Übungsaufgaben: 4diac Trainingslösungen (4.2)

Franz Höpfinger

Inhaltsverzeichnis

1 Übungsaufgaben: 4diac Trainingslösungen 5

2 Uebungen 7

2.1 test AX 7

2.1.1 Meins 7

2.1.2 Numerischer Index (test_AX) 81

2.2 test B 886

2.2.1 Numerischer Index (test_B) 886

2.2.2 Numerischer Index (Versuche, test_B) 1427

2.3 test VV 1440

2.3.1 01 Multicast 1440

2.3.2 02 RequestReply 1442

2.3.3 03 OPC UA 1446

2.3.4 Numerischer Index (test_VV) 1468

3 static 1471

3.1 Fonts 1471

3.1.1 TGL Normschrift – _std Font Variants 1471

1. Übungsaufgaben: 4diac Trainingslösungen

Dieses Wiki (**4.2**) enthält die Musterlösungen zu den Übungsaufgaben (`test_AX`, `test_B`, `test_VV`) für Eclipse 4diac, ausgelagert aus dem Haupt-Wiki Visuelle Programmiersprachen: Low-Code (4).

Siehe auch: Wiki 4.1 – 4diac Bibliotheken: FB- & Adapter-Referenz

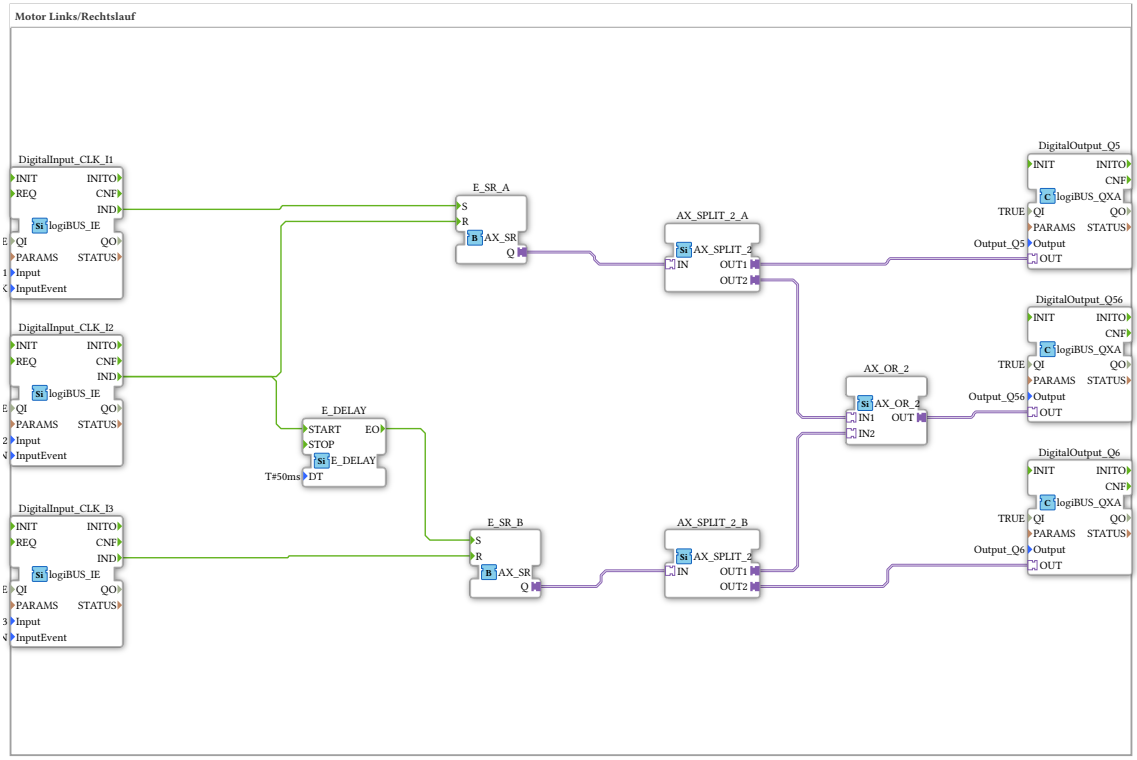
- PDF-Handbuch herunterladen

2. Uebungen

2.1 test AX

2.1.1 Meins

2.1.1.1 ErsteAutomatisierung: Motor Links-/Rechtslauf



• — ○ • — ○ • — ○ •

Einleitung

`ErsteAutomatisierung` ist eine frühe Übung (Kommentar im Typ: „Motor Links/Rechtslauf“). Aus drei Tastern (`Input_I1`…`Input_I3`) wird über zwei Set/Reset-Speicher die Ansteuerung zweier Ausgänge (`Output_Q5`, `Output_Q6`) für die beiden Drehrichtungen gebildet; ein dritter Ausgang (`Output_Q56`) ist die ODER-Verknüpfung beider Richtungen. Die SubApp hat keine eigene Schnittstelle.

Verwendete Funktionsbausteine (FBs)

Instanz	Typ	Parameter	Erklärung
`DigitalInput_CLK_I1`	`logiBUS::io::DI::logiBUS_IE`	`Input = Input_I1`, `InputEvent = BUTTON_SINGLE_CLICK`, `QI = TRUE`	Liefert bei Einzelklick auf Taster I1 das Ereignis `IND`.
`DigitalInput_CLK_I2`	`logiBUS::io::DI::logiBUS_IE`	`Input = Input_I2`, `InputEvent = BUTTON_PRESS_DOWN`, `QI = TRUE`	Liefert bei Niederdrücken von Taster I2 das Ereignis `IND`.

DigitalInput_CLK_I3	logiBUS::io::DI::logiBUS_DI	Input = Input_I3, InputEvent = BUTTON_PRESS_DOWN, QI = TRUE	Liefert bei Niederdrücken von Taster I3 das Ereignis IND.
E_DELAY	iec61499::events::E_DELAY	DT = T#50ms	Verzögert das Ereignis von I2 um 50 ms.
E_SR_A	adapter::events::unidirectional::AX_SR	-	Set/Reset-Speicher für Richtung A (Ausgang Q).
E_SR_B	adapter::events::unidirectional::AX_SR	-	Set/Reset-Speicher für Richtung B (Ausgang Q).
AX_SPLIT_2_A	adapter::events::unidirectional::AX_SPLIT_2	-	Verteilt den Zustand von E_SR_A auf zwei Verbraucher.
AX_SPLIT_2_B	adapter::events::unidirectional::AX_SPLIT_2	-	Verteilt den Zustand von E_SR_B auf zwei Verbraucher.
AX_OR_2	adapter::booleanOperators::AX_OR_2	-	ODER-Verknüpfung der beiden Richtungen.
DigitalOutput_Q5	logiBUS::io::DQ::logiBUS_DQ	Output = Output_Q5, QI = TRUE	Ausgang für Richtung A.
DigitalOutput_Q6	logiBUS::io::DQ::logiBUS_DQ	Output = Output_Q6, QI = TRUE	Ausgang für Richtung B.
DigitalOutput_Q56	logiBUS::io::DQ::logiBUS_DQ	Output = Output_Q56, QI = TRUE	Gemeinsamer Ausgang: aktiv, wenn A oder B aktiv ist.

Programmablauf und Verbindungen

Ereignisverbindungen:

1. `DigitalInput_CLK_I1.IND` ⟶ `E_SR_A.S`: Einzelklick auf I1 setzt Richtung A.
2. `DigitalInput_CLK_I2.IND` ⟶ `E_SR_A.R`: Niederdrücken von I2 setzt Richtung A zurück.
3. `DigitalInput_CLK_I2.IND` ⟶ `E_DELAY.START` und `E_DELAY.EO` ⟶ `E_SR_B.S`: dasselbe Ereignis setzt nach 50 ms Richtung B.
4. `DigitalInput_CLK_I3.IND` ⟶ `E_SR_B.R`: Niederdrücken von I3 setzt Richtung B zurück.

Adapterverbindungen:

1. `E_SR_A.Q` ⟶ `AX_SPLIT_2_A.IN`; `AX_SPLIT_2_A.OUT1` ⟶ `DigitalOutput_Q5.OUT`; `AX_SPLIT_2_A.OUT2` ⟶ `AX_OR_2.IN1`.
2. `E_SR_B.Q` ⟶ `AX_SPLIT_2_B.IN`; `AX_SPLIT_2_B.OUT2` ⟶ `DigitalOutput_Q6.OUT`; `AX_SPLIT_2_B.OUT1` ⟶ `AX_OR_2.IN2`.
3. `AX_OR_2.OUT` ⟶ `DigitalOutput_Q56.OUT`.

Ein Zurücksetzen von Richtung B beim Setzen von A (I1) ist im Netzwerk nicht verschaltet. Beim Umschalten über I2 wird A zurückgesetzt und B erst 50 ms später gesetzt.

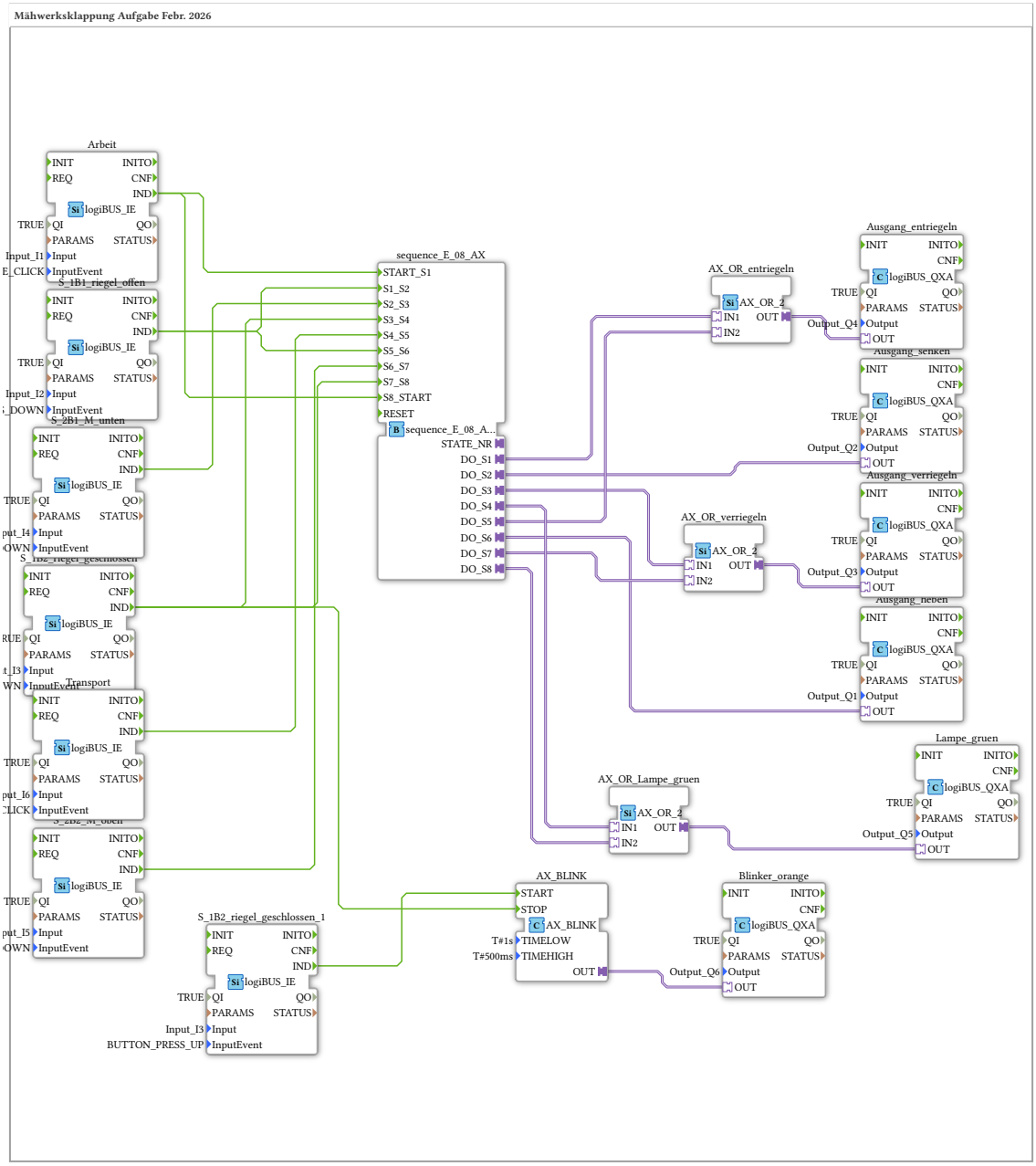
Zusammenfassung

`ErsteAutomatisierung` zeigt mit drei Tastern, zwei Set/Reset-Speichern, einer 50-ms-Verzögerung und einer ODER-Verknüpfung, wie zwei Ausgänge für die Drehrichtungen eines Motors sowie ein gemeinsamer Ausgang aufgebaut werden.

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

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

2.1.1.2 Maehwerksklappung: Mähwerksklappung (Ablaufkette)



• – ○ • – ○ • – ○ •

Einleitung

`Maehwerksklappung` ist die Trainingsaufgabe „Mähwerksklappung“ (Stand Februar 2026): Ein Mähwerk wird über einen Riegel (Entriegeln/Verriegeln) und einen Zylinder (Heben/Senken) zwischen den Zuständen **Arbeit** (unten) und **Transport** (oben) umgeklappt. Die Übung ist als **8-stufige Ablaufkette** mit `sequence_E_08_AX` aufgebaut; die Schritte werden durch Tastereignisse (`logiBUS_IE`) weitergeschaltet. Die Variante mit Pegel-Eingängen und reiner Boolescher Logik ist in `Maehwerksklappung_IXA` beschrieben.

Die Schnittstelle der SubApp ist leer, alle Ein- und Ausgänge sind direkt auf `Input_I1`…`Input_I6` und `Output_Q1`…`Output_Q6` gelegt.

Verwendete Funktionsbausteine (FBs)

Eingänge (Ereignisse)

Instanz	Typ	Parameter	Erklärung
Arbeit	logiBUS_IE	Input_I1, BUTTON_SINGLE_CLICK	Taster „Arbeit“ (Einzelklick)
S_1B1_riegel_offen	logiBUS_IE	Input_I2, BUTTON_PRESS_DOWN	Sensor 1B1: Riegel offen (Flanke beim Drücken)
S_1B2_riegel_geschlossen	logiBUS_IE	Input_I3, BUTTON_PRESS_DOWN	Sensor 1B2: Riegel geschlossen (Flanke beim Drücken)
S_1B2_riegel_geschlossen_1	logiBUS_IE	Input_I3, BUTTON_PRESS_UP	Derselbe Sensor 1B2, aber Flanke beim Loslassen
S_2B1_M_unten	logiBUS_IE	Input_I4, BUTTON_PRESS_DOWN	Sensor 2B1: Mähwerk unten
S_2B2_M_oben	logiBUS_IE	Input_I5, BUTTON_PRESS_DOWN	Sensor 2B2: Mähwerk oben
Transport	logiBUS_IE	Input_I6, BUTTON_SINGLE_CLICK	Taster „Transport“ (Einzelklick)

Ablaufsteuerung und Logik

Instanz	Typ	Parameter	Erklärung
sequence_E_08_AX	sequence_E_08_AX	–	Ereignisgesteuerte Schrittkette mit 8 Zuständen; jeder Zustand aktiviert einen Adapter-Ausgang DO_S1…DO_S8
AX_OR_entriegeln	AX_OR_2	–	ODER aus DO_S1 und DO_S5 für den Ausgang „entriegeln“
AX_OR_verriegeln	AX_OR_2	–	ODER aus DO_S3 und DO_S7 für den Ausgang „verriegeln“
AX_OR_Lampe_gruen	AX_OR_2	–	ODER aus DO_S4 und DO_S8 für die grüne Lampe
AX_BLINK	AX_BLINK	TIMELOW = T#1s, TIMEHIGH = T#500ms	Blinkgeber für die orange Warnleuchte

Ausgänge

Instanz	Typ	Parameter	Erklärung
Ausgang_heben	logiBUS_QXA	Output_Q1	Mähwerk heben
Ausgang_senken	logiBUS_QXA	Output_Q2	Mähwerk senken
Ausgang_verriegeln	logiBUS_QXA	Output_Q3	Riegel schließen
Ausgang_entriegeln	logiBUS_QXA	Output_Q4	Riegel öffnen
Lampe_gruen	logiBUS_QXA	Output_Q5	Grüne Lampe

Blinker_orange	logiBUS_QXA	Output_Q6	Orange Blinkleuchte

Alle Instanzen von logiBUS_QXA, logiBUS_IE setzen QI auf TRUE.

Programmablauf und Verbindungen

Im Diagramm sind zwei Kommentare vermerkt: „S4 ist der Zustand ‚Arbeit'" und „S8 ist Zustand ‚Transport'". Die Schrittkette läuft wie folgt:

Schritt	Weiterschaltung durch (Ereignis)	Aktiver Ausgang im Zustand
Start ⟶ S1	Arbeit.IND (START_S1)	S1: DO_S1 ⟶ entriegeln (Q4)
S1 ⟶ S2	S_1B1_riegel_offen.IND	S2: DO_S2 ⟶ senken (Q2)
S2 ⟶ S3	S_2B1_M_unten.IND	S3: DO_S3 ⟶ verriegeln (Q3)
S3 ⟶ S4	S_1B2_riegel_geschlossen.IND	S4: DO_S4 ⟶ grüne Lampe (Q5), Zustand „Arbeit"
S4 ⟶ S5	Transport.IND	S5: DO_S5 ⟶ entriegeln (Q4)
S5 ⟶ S6	S_1B1_riegel_offen.IND	S6: DO_S6 ⟶ heben (Q1)
S6 ⟶ S7	S_2B2_M_oben.IND	S7: DO_S7 ⟶ verriegeln (Q3)
S7 ⟶ S8	S_1B2_riegel_geschlossen.IND	S8: DO_S8 ⟶ grüne Lampe (Q5), Zustand „Transport"
S8 ⟶ Start	Arbeit.IND (S8_START)	–

- Entriegeln (Q4) und Verriegeln (Q3) werden jeweils in zwei Zuständen benötigt und deshalb über AX_OR_entriegeln (S1, S5) und AX_OR_verriegeln (S3, S7) zusammengeführt. Dasselbe gilt für die grüne Lampe (S4, S8) über AX_OR_Lampe_gruen. Senken (S2) und Heben (S6) gehen direkt auf ihre Ausgänge.
- **Warnblinker**: Das Ereignis S_1B2_riegel_geschlossen.IND (Sensor 1B2 gedrückt) stoppt AX_BLINK (STOP), das Loslassen von 1B2 (S_1B2_riegel_geschlossen_1.IND) startet ihn (START). Der Blinkausgang AX_BLINK.OUT geht auf Blinker_orange (Q6). Damit blinkt die orange Leuchte, solange der Riegel nicht als geschlossen gemeldet wird.

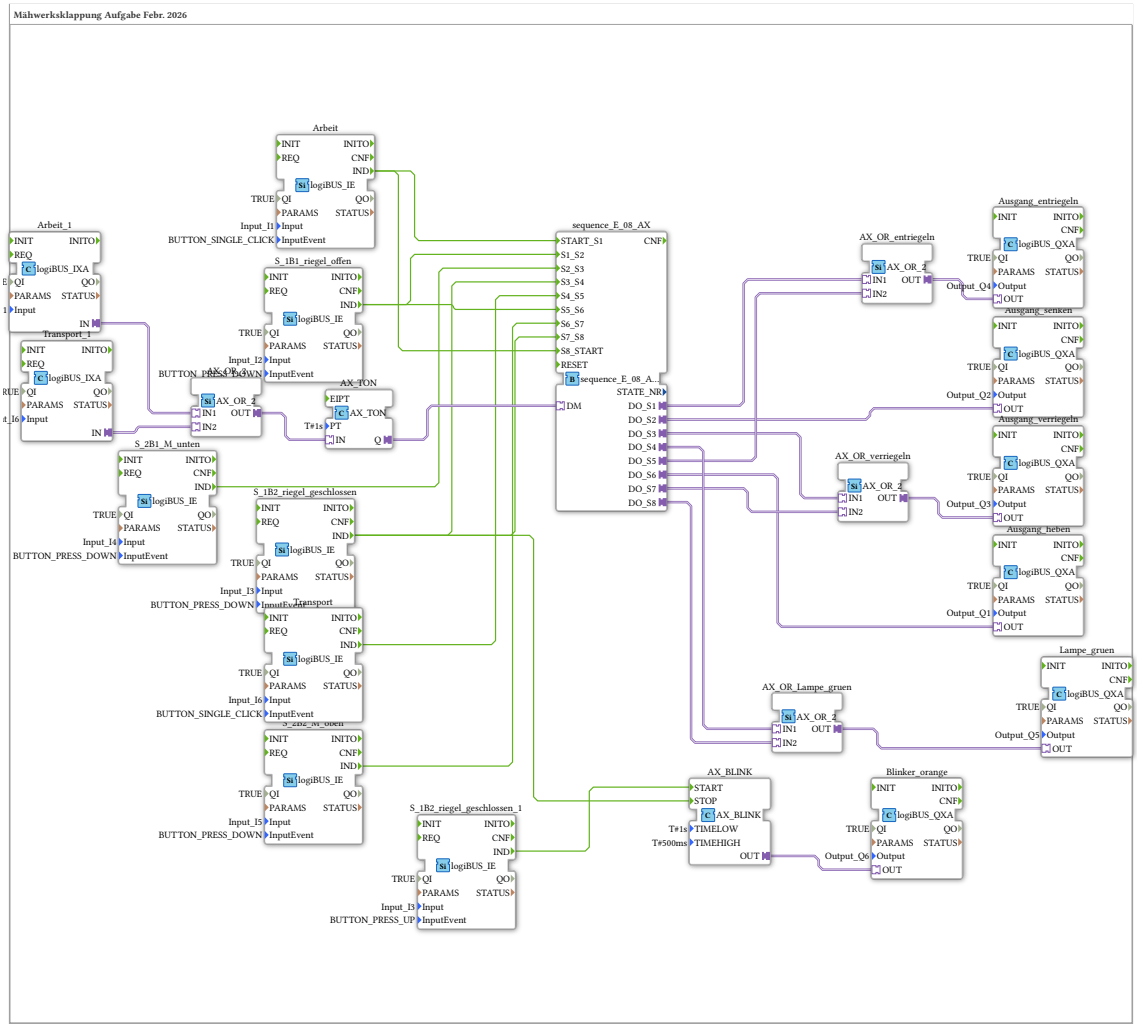
Zusammenfassung

Maehwerksklappung löst die Mähwerks-Aufgabe als klassische ereignisgesteuerte Schrittkette: Acht Zustände, durch Taster- und Sensorereignisse weitergeschaltet, steuern Riegel, Zylinder und Leuchten; ein separater Blinkgeber folgt allein dem Sensor 1B2.

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

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

2.1.1.3 Maehwerksklappung_DM: Mähwerksklappung mit Totmann



• – ○ • – ○ • – ○ •

Einleitung

`Maehwerksklappung_DM` ist die Aufgabe „Mähwerksklappung“ vom Februar 2026 (Kommentar im Typ) in der Variante mit Totmannschalter (`DM`). Eine achtstufige Ablaufsteuerung `sequence_E_08_AX_DM` schaltet über sechs Eingänge (`Input_I1`…`Input_I6`) die Ausgänge `Output_Q1`…`Output_Q6` für Heben, Senken, Verriegeln, Entriegeln, grüne Lampe und orange Blinklampe. Die SubApp hat keine eigene Schnittstelle.

Verwendete Funktionsbausteine (FBs)

Instanz	Typ	Parameter	Erklärung
`Arbeit`	`logiBUS::io::DI::logiBUS_IE`	`Input = Input_I1, InputEvent = BUTTON_SINGLE_CLICK`	Taster Arbeit: startet bzw. beendet die Sequenz.
`S_1B1_riegel_offen`	`logiBUS::io::DI::logiBUS_IE`	`Input = Input_I2, InputEvent = BUTTON_PRESS_DOWN`	Meldung Riegel offen.
`S_1B2_riegel_geschlossen`	`logiBUS::io::DI::logiBUS_IE`	`Input = Input_I3, InputEvent = BUTTON_PRESS_DOWN`	Meldung Riegel geschlossen.

S_1B2_riegel_geschlossen	logiBUS::io::DI::logiBUS_IX	Input = Input_I3, InputEvent = BUTTON_PRESS_UP	Gleicher Eingang I3, aber Ereignis beim Loslassen.
S_2B1_M_unten	logiBUS::io::DI::logiBUS_IX	Input = Input_I4, InputEvent = BUTTON_PRESS_DOWN	Meldung Mähwerk unten.
S_2B2_M_oben	logiBUS::io::DI::logiBUS_IX	Input = Input_I5, InputEvent = BUTTON_PRESS_DOWN	Meldung Mähwerk oben.
Transport	logiBUS::io::DI::logiBUS_IX	Input = Input_I6, InputEvent = BUTTON_SINGLE_CLICK	Taster Transport.
Arbeit_1	logiBUS::io::DI::logiBUS_IXA	Input = Input_I1	Pegel von I1 als Adapter (für den Totmann-Zweig).
Transport_1	logiBUS::io::DI::logiBUS_IXA	Input = Input_I6	Pegel von I6 als Adapter (für den Totmann-Zweig).
AX_OR_2	adapter::booleanOperators::AX_OR_2	-	Verknüpft Arbeit_1 und Transport_1.
AX_TON	adapter::events::unidirectional::timers::AX_TON	PT = T#1s	Einschaltverzögerung 1 s; Ausgang Q speist den Eingang DM der Sequenz.
sequence_E_08_AX	logiBUS::unidirectionals::sequence::event::sequence_E_08_AX_DM		Achtstufige ereignisgesteuerte Ablaufsteuerung mit Totmann-Eingang (DM); Ausgänge DO_S1…DO_S8.
AX_OR_entriegeln	adapter::booleanOperators::AX_OR_2	-	DO_S1 oder DO_S5 → Ausgang entriegeln.
AX_OR_verriegeln	adapter::booleanOperators::AX_OR_2	-	DO_S3 oder DO_S7 → Ausgang verriegeln.
AX_OR_Lampe_gruen	adapter::booleanOperators::AX_OR_2	-	DO_S4 oder DO_S8 → grüne Lampe.
AX_BLINK	adapter::events::unidirectional::signalGenerators::AX_BLINK	TIMELOW = T#1s, TIMEHIGH = T#500ms	Blinkgenerator für die orange Lampe.
Ausgang_heben	logiBUS::io::DQ::logiBUS_QXA	Output = Output_Q1	Mähwerk heben.
Ausgang_senken	logiBUS::io::DQ::logiBUS_QXA	Output = Output_Q2	Mähwerk senken.
Ausgang_verriegeln	logiBUS::io::DQ::logiBUS_QXA	Output = Output_Q3	Verriegeln.
Ausgang_entriegeln	logiBUS::io::DQ::logiBUS_QXA	Output = Output_Q4	Entriegeln.
Lampe_gruen	logiBUS::io::DQ::logiBUS_QXA	Output = Output_Q5	Grüne Lampe.
Blinker_orange	logiBUS::io::DQ::logiBUS_QXA	Output = Output_Q6	Orange Blinklampe.

Programmablauf und Verbindungen

Schrittkette (Ereignisse → `sequence_E_08_AX`):

Übergang	Auslösendes Ereignis
START_S1	Arbeit.IND
S1_S2	S_1B1_riegel_offen.IND
S2_S3	S_2B1_M_unten.IND
S3_S4	S_1B2_riegel_geschlossen.IND
S4_S5	Transport.IND
S5_S6	S_1B1_riegel_offen.IND
S6_S7	S_2B2_M_oben.IND
S7_S8	S_1B2_riegel_geschlossen.IND
S8_START	Arbeit.IND

Die Kommentare im Netzwerk benennen S4 als Zustand „Arbeit“ und S8 als Zustand „Transport“.

Zuordnung der Zustandsausgänge:

1. DO_S1 und DO_S5 ⟶ AX_OR_entriegeln ⟶ Ausgang_entriegeln (Q4).
2. DO_S2 ⟶ Ausgang_senken (Q2).
3. DO_S3 und DO_S7 ⟶ AX_OR_verriegeln ⟶ Ausgang_verriegeln (Q3).
4. DO_S4 und DO_S8 ⟶ AX_OR_Lampe_gruen ⟶ Lampe_gruen (Q5).
5. DO_S6 ⟶ Ausgang_heben (Q1).

Blinker: S_1B2_riegel_geschlossen.IND (I3 gedrückt) ⟶ AX_BLINK.STOP; S_1B2_riegel_geschlossen_1.IND (I3 losgelassen) ⟶ AX_BLINK.START; AX_BLINK.OUT ⟶ Blinker_orange (Q6).

Totmann: Arbeit_1.IN und Transport_1.IN ⟶ AX_OR_2 ⟶ AX_TON (1 s) ⟶ sequence_E_08_AX.DM.

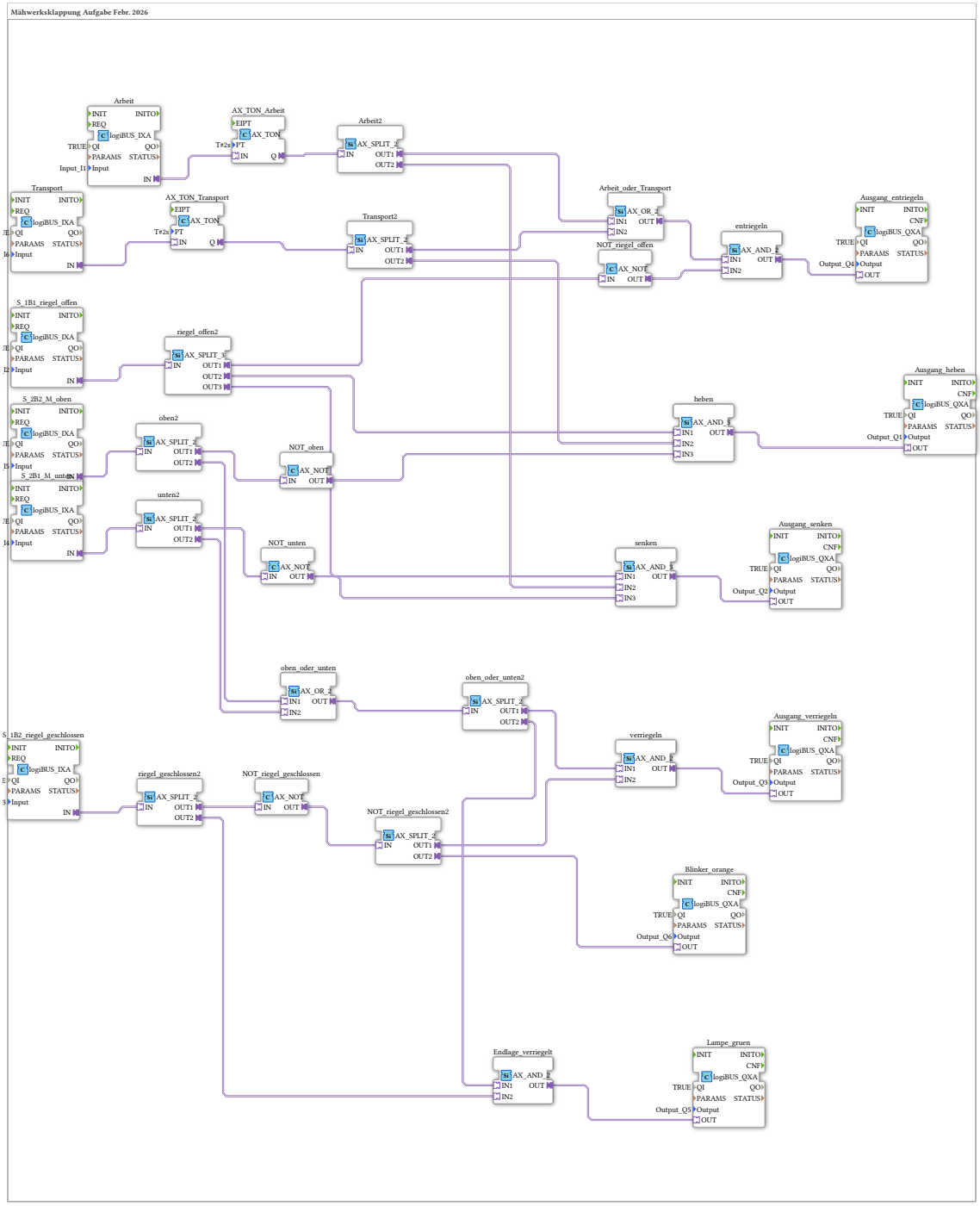
Zusammenfassung

Maehwerksklappung_DM zeigt eine vollständig ereignisgesteuerte achtstufige Sequenz (Entriegeln, Senken, Verriegeln, Arbeit, Entriegeln, Heben, Verriegeln, Transport) mit Totmann-Freigabe über eine 1-s-Einschaltverzögerung sowie einer Blinkanzeige.

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

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

2.1.1.4 Maehwerksklappung_IXA: Mähwerksklappung (Boolesche Logik)



• — ○ • — ○ • — ○ •

Einleitung

`Maehwerksklappung_IXA` löst dieselbe Aufgabe „Mähwerksklappung“ (Stand Februar 2026) wie `Maehwerksklappung`, aber ohne Schrittkette: Die Eingänge werden als **Pegel** (`logiBUS_IXA`) gelesen und die Ausgänge ausschließlich durch Boolesche Verknüpfungen (UND/ODER/NICHT) über Adapter gebildet. Die Logikgleichungen stehen als Kommentare im Netzwerk (Y_1 … Y_4, H_1, H_2).

Die Schnittstelle der SubApp ist leer; Eingänge `Input_I1`…`Input_I6`, Ausgänge `Output_Q1`…`Output_Q6`.

Verwendete Funktionsbausteine (FBs)

Eingänge

Instanz	Typ	Parameter	Erklärung
`Arbeit`	logiBUS_IXA	`Input_I1`	Taster „Arbeit“
`S_1B1_riegel_offen`	logiBUS_IXA	`Input_I2`	Sensor 1B1: Riegel offen
`S_1B2_riegel_geschlossen`	logiBUS_IXA	`Input_I3`	Sensor 1B2: Riegel geschlossen
`S_2B1_M_unten`	logiBUS_IXA	`Input_I4`	Sensor 2B1: Mähwerk unten
`S_2B2_M_oben`	logiBUS_IXA	`Input_I5`	Sensor 2B2: Mähwerk oben
`Transport`	logiBUS_IXA	`Input_I6`	Taster „Transport“
`AX_TON_Arbeit`, `AX_TON_Transport`	AX_TON	`PT = T#2s`	Einschaltverzögerung 2 s für die Taster „Arbeit“ bzw. „Transport“

Logik

Instanz	Typ	Erklärung
`riegel_offen2`	AX_SPLIT_3	Verteilt 1B1 auf drei Verbraucher (Entriegeln, Heben, Senken)
`Arbeit2`, `Transport2`, `oben2`, `unten2`, `riegel_geschlossen2`, `oben_oder_unten2`, `NOT_riegel_geschlossen2`	AX_SPLIT_2	Verteilen ein Signal auf zwei Verbraucher
`NOT_riegel_offen`, `NOT_oben`, `NOT_unten`, `NOT_riegel_geschlossen`	AX_NOT	Negation von 1B1, 2B2, 2B1 bzw. 1B2
`Arbeit_oder_Transport`, `oben_oder_unten`	AX_OR_2	ODER aus „Arbeit“/„Transport“ bzw. „oben“/„unten“
`entriegeln`, `verriegeln`, `Endlage_verriegelt`	AX_AND_2	UND-Verknüpfung für Entriegeln, Verriegeln und Statuslampe
`heben`, `senken`	AX_AND_3	UND-Verknüpfung mit drei Eingängen für Heben und Senken

Ausgänge

Instanz	Typ	Parameter	Erklärung
`Ausgang_heben`	logiBUS_QXA	`Output_Q1`	Mähwerk heben
`Ausgang_senken`	logiBUS_QXA	`Output_Q2`	Mähwerk senken
`Ausgang_verriegeln`	logiBUS_QXA	`Output_Q3`	Riegel schließen
`Ausgang_entriegeln`	logiBUS_QXA	`Output_Q4`	Riegel öffnen
`Lampe_gruen`	logiBUS_QXA	`Output_Q5`	Statusleuchte H_2
`Blinker_orange`	logiBUS_QXA	`Output_Q6`	Warnleuchte H_1

Programmablauf und Verbindungen

Die Logik entspricht den Kommentaren im Netzwerk (S_1 = „Transport“, S_2 = „Arbeit“; 1B1 Riegel offen, 1B2 Riegel geschlossen, 2B1 unten, 2B2 oben):

1. **Entriegeln** (Q4): `Y_1 = (S_1 oder S_2) und nicht 1B1`. Hat der Fahrer „Arbeit“ oder „Transport“ gedrückt und der Riegel ist noch nicht offen, wird der Riegel geöffnet (`Arbeit_oder_Transport` ⟶ `entriegeln` mit `NOT_riegel_offen`).
2. **Heben** (Q1): `Y_3 = S_1 und 1B1 und nicht 2B2`. Aktiv, solange „Transport“ gehalten wird, der Riegel offen ist und die obere Endlage noch nicht erreicht ist (`heben`). Beim Loslassen fällt der Ausgang ab, das Ventil stoppt den Zylinder.
3. **Senken** (Q2): `Y_4 = S_2 und 1B1 und nicht 2B1`. Analog zum Heben mit „Arbeit“ und der unteren Endlage (`senken`).
4. **Verriegeln** (Q3): `Y_2 = (2B2 oder 2B1) und nicht 1B2`. Sobald das Mähwerk in einer Endlage ist und der Riegel noch nicht geschlossen ist, wird verriegelt (`oben_oder_unten` ⟶ `verriegeln`).
5. **Warnleuchte** (Q6): `H_1 = nicht 1B2`. Leuchtet, solange der Riegel nicht komplett geschlossen ist (`NOT_riegel_geschlossen` ⟶ `Blinker_orange`).
6. **Statusleuchte** (Q5): `H_2 = 1B2 und (2B2 oder 2B1)`. Signalisiert den sicheren Zustand „verriegelt in Endlage“ (`Endlage_verriegelt` ⟶ `Lampe_gruen`).

Die Taster „Arbeit“ und „Transport“ laufen jeweils über eine 2-s-Einschaltverzögerung (`AX_TON_Arbeit`, `AX_TON_Transport`), bevor sie in die Gleichungen für Entriegeln, Heben und Senken eingehen; in den Kommentaren des Netzwerks ist diese Verzögerung nicht erwähnt.

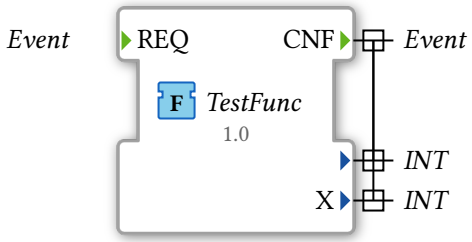
Zusammenfassung

`Maehwerksklappung_IXA` setzt die Mähwerks-Aufgabe ohne Zustandsspeicher als reine Kombinatorik um: Sechs Eingänge werden mit Splittern verteilt, mit UND/ODER/NICHT verknüpft und steuern Riegel, Zylinder und beide Leuchten direkt aus dem aktuellen Sensorbild.

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

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

2.1.1.5 TestFunc: Funktion (IEC 61499 Function)



• – ◦ • – ◦ • – ◦ •

Einleitung

`TestFunc` ist keine SubApp, sondern ein Funktionstyp (`.fct`, Standard 61499-1) im Paket `Meins` mit dem Kommentar „Function with empty body“. Er dient als einfaches Testobjekt für die Funktions-Syntax: ein Ereigniseingang `REQ`, ein Ereignisausgang `CNF` und ein Strukturierter-Text-Rumpf. Es gibt keine Funktionsbaustein-Instanzen und keine Verbindungen.

Schnittstelle der Funktion

Instanz	Typ	Parameter	Erklärung
`REQ`	`Event (Ereigniseingang)`	-	Anforderung zur Ausführung.
`CNF`	`Event (Ereignisausgang)`	`With = X` (und eine unbenannte Variable)	Bestätigung; liefert die beiden Ausgangswerte.
`X`	`INT (Ausgangsvariable)`	Initialwert `5`	Ausgabeparameter, im Rumpf auf 7 gesetzt.

(unbenannt)	`INT (Ausgangsvariable)`	-	Zweite Ausgangsvariable mit leerem Namen; entspricht dem Rückgabewert `TestFunc`.

Programmablauf und Verbindungen

Der Rumpf ist in Structured Text geschrieben:

```
PACKAGE Meins;

FUNCTION TestFunc : INT

VAR_OUTPUT
    X : INT := 5;
END_VAR

TestFunc := 5;
X := 7;
END_FUNCTION
```

1. Der Rückgabewert `TestFunc` (INT) wird auf `5` gesetzt.
2. Der Ausgabeparameter `X` (Initialwert `5`) wird auf `7` gesetzt.

Obwohl der Kommentar „empty body“ lautet, enthält der Rumpf diese beiden Zuweisungen.

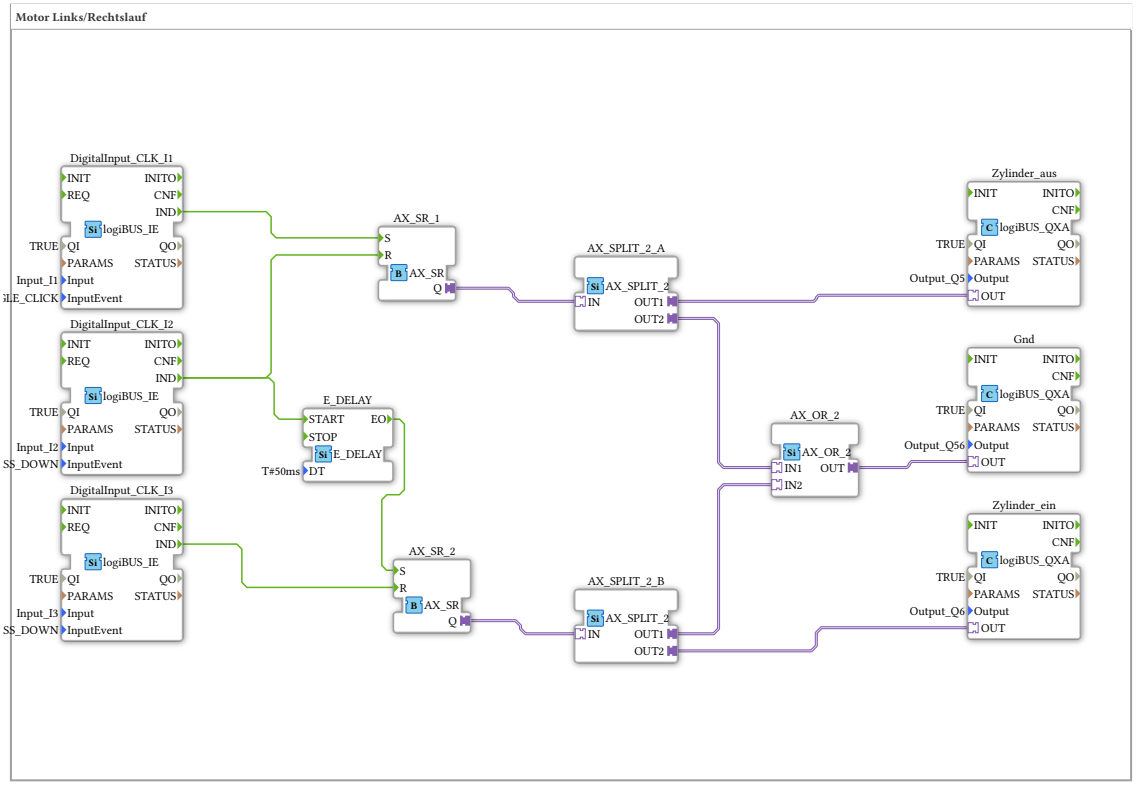
Zusammenfassung

`TestFunc` ist ein minimales Beispiel einer IEC-61499-Funktion mit Ereignis-Schnittstelle, Rückgabewert und einem Ausgabeparameter.

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

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

2.1.1.6 ZylinderAusEinAutomatik: Zylinder aus/ein per Taster



• — ○ • — ○ • — ○ •

Einleitung

ZylinderAusEinAutomatik steuert einen Zylinder, der über zwei Ausgänge (Output_Q5 = ausfahren, Output_Q6 = einfahren) und einen gemeinsamen Masse-Ausgang (Output_Q56) angeschlossen ist. Drei Taster schalten die Bewegung: Einer startet das Ausfahren, einer stoppt es und startet nach kurzer Verzögerung das Einfahren, einer beendet das Einfahren. Die Schnittstelle der SubApp ist leer. (Der Comment der SubApp lautet noch „Motor Links/Rechtslauf“ und stammt offenbar von einer Vorlage.)

Ein Kommentar im Netzwerk vermerkt: „Der Zylinder macht beim Ausfahren ein summendes Geräusch.“

Verwendete Funktionsbausteine (FBs)

Instanz	Typ	Parameter	Erklärung
DigitalInput_CLK_I1	logiBUS_IE	Input_I1, BUTTON_SINGLE_CLICK, QI = TRUE	Taster „Ausfahren“ (Einzelklick)
DigitalInput_CLK_I2	logiBUS_IE	Input_I2, BUTTON_PRESS_DOWN, QI = TRUE	Taster „Einfahren starten“ (beim Drücken)
DigitalInput_CLK_I3	logiBUS_IE	Input_I3, BUTTON_PRESS_DOWN, QI = TRUE	Taster „Einfahren beenden“ (beim Drücken)
AX_SR_1	AX_SR	–	Speicher (Set/Reset) für „Zylinder ausfahren“
AX_SR_2	AX_SR	–	Speicher (Set/Reset) für „Zylinder einfahren“
E_DELAY	E_DELAY	DT = T#50ms	Verzögert den Start des Einfahrens um 50 ms
AX_SPLIT_2_A, AX_SPLIT_2_B	AX_SPLIT_2	–	Verteilen jeweils einen Speicherzustand auf den Zylinderausgang und das ODER
AX_OR_2	AX_OR_2	–	ODER der beiden Bewegungen für den gemeinsamen Ausgang
Zylinder_aus	logiBUS_QXA	Output_Q5, QI = TRUE	Zylinder ausfahren
Zylinder_ein	logiBUS_QXA	Output_Q6, QI = TRUE	Zylinder einfahren
Gnd	logiBUS_QXA	Output_Q56, QI = TRUE	Gemeinsamer Ausgang, aktiv wenn eine der beiden Bewegungen läuft

Programmablauf und Verbindungen

1. **Ausfahren starten**: DigitalInput_CLK_I1.IND setzt AX_SR_1 (S). AX_SR_1.Q geht über AX_SPLIT_2_A auf Zylinder_aus (Q5) und auf AX_OR_2.IN1.
2. **Ausfahren stoppen, Einfahren starten**: DigitalInput_CLK_I2.IND setzt AX_SR_1 zurück (R) und startet E_DELAY. Nach 50 ms setzt E_DELAY.EO den Speicher AX_SR_2 (S). AX_SR_2.Q geht über AX_SPLIT_2_B auf Zylinder_ein (Q6) und auf AX_OR_2.IN2.
3. **Einfahren beenden**: DigitalInput_CLK_I3.IND setzt AX_SR_2 zurück (R).
4. **Gemeinsamer Ausgang**: AX_OR_2.OUT steuert Gnd (Q56) an; dieser ist aktiv, sobald entweder Ausfahren oder Einfahren aktiv ist.

Die 50 ms Verzögerung (E_DELAY) liegt zwischen dem Zurücksetzen von AX_SR_1 und dem Setzen von AX_SR_2.

Zusammenfassung

`ZylinderAusEinAutomatik` zeigt eine Zylindersteuerung mit zwei Set/Reset-Speichern, einer kurzen Umschaltverzögerung und einem gemeinsam angesteuerten Masse-Ausgang.

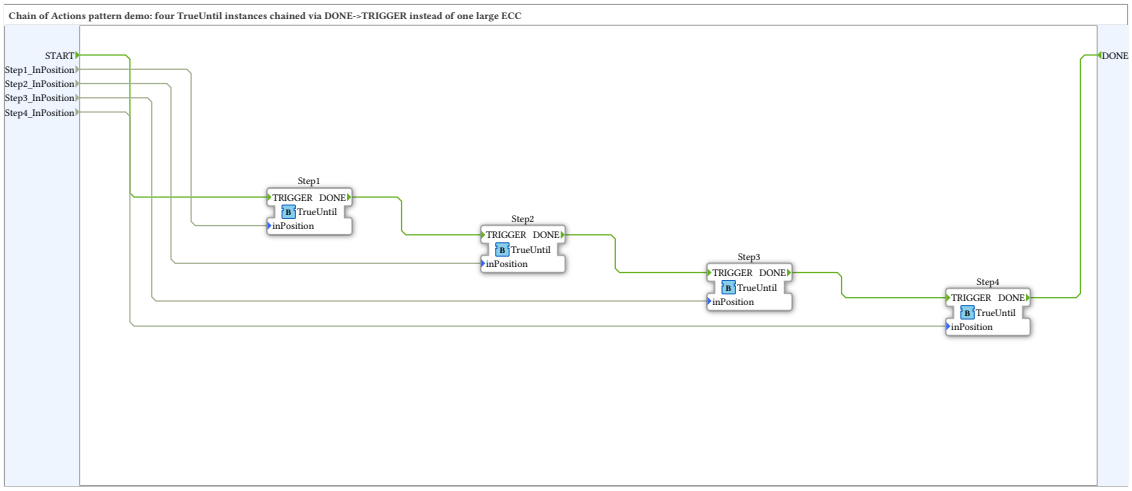
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

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

2.1.1.7 DesingPatterns

ChainOfActionsPattern

ChainOfActionsDemo: Chain-of-Actions-Muster mit vier TrueUntil



• – ○ • – ○ • – ○ •

Einleitung

`ChainOfActionsDemo` zeigt das Chain-of-Actions-Muster: vier Instanzen desselben Bausteins `TrueUntil` werden über `DONE` ⟶ `TRIGGER` hintereinander gekettet, statt einen großen Ablaufautomaten (ECC) zu verwenden (Kommentar im Typ). Schnittstelle: Ereigniseingang `START`, Ereignisausgang `DONE` sowie vier BOOL-Eingänge `Step1_InPosition`…`Step4_InPosition`, mit denen im Demo die Rückmeldung „Position erreicht“ der jeweiligen Aktorik simuliert wird.

Verwendete Funktionsbausteine (FBs)

Instanz	Typ	Parameter	Erklärung
`Step1`	`Meins::DesingPatterns::ChainOfActionsPattern::TrueUntil`	-	Schritt 1 der Kette.
`Step2`	`Meins::DesingPatterns::ChainOfActionsPattern::TrueUntil`	-	Schritt 2 der Kette.
`Step3`	`Meins::DesingPatterns::ChainOfActionsPattern::TrueUntil`	-	Schritt 3 der Kette.
`Step4`	`Meins::DesingPatterns::ChainOfActionsPattern::TrueUntil`	-	Schritt 4 der Kette.

Programmablauf und Verbindungen

`TrueUntil` ist ein Basis-FB (im Projekt unter `DesingPatterns/ChainOfActionsPattern/TrueUntil.fbt`). Laut seiner ECC wartet er in `START` auf `TRIGGER` (oder `REQ`), gibt im Zustand `MOVING` das Ereignis `TO_POSITION` aus und wartet auf `inPosition`; danach gibt er `STOP` und `DONE` aus.

Ereignisverbindungen:

1. `START` ⟶ `Step1.TRIGGER`
2. `Step1.DONE` ⟶ `Step2.TRIGGER`
3. `Step2.DONE` ⟶ `Step3.TRIGGER`
4. `Step3.DONE` ⟶ `Step4.TRIGGER`
5. `Step4.DONE` ⟶ `DONE`

Datenverbindungen: `Step1_InPosition` ⟶ `Step1.inPosition`, entsprechend `Step2_InPosition` bis `Step4_InPosition` zu den jeweiligen Schritten.

Die Ausgänge `TO_POSITION` und `STOP` der Schritte sind in diesem Demo nicht verschaltet; sie wären mit der konkreten Aktorik zu verbinden.

Zusammenfassung

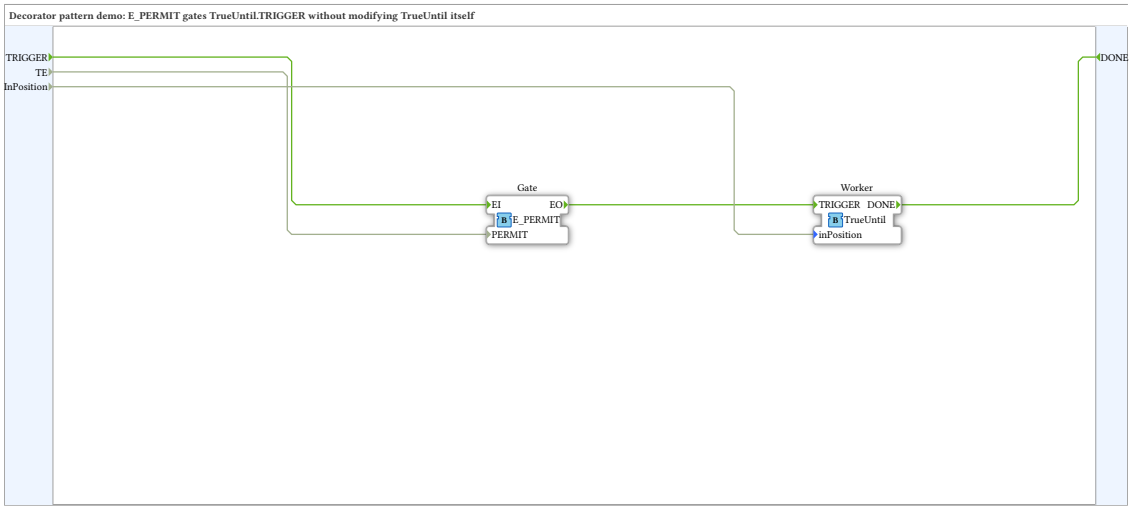
Das Demo setzt eine Bewegungsfolge aus vier identischen, wiederverwendbaren Schritt-Bausteinen zusammen, die nur über `DONE` ⟶ `TRIGGER` verkettet sind.

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

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

DecoratorPattern

DecoratorDemo: Decorator-Muster (Freigabe-Tor)



• – ○ • – ○ • – ○ •

Einleitung

`DecoratorDemo` ist die Demo zum **Decorator-Muster** (IEC-61499-Grundkurs, Modul 6, Valeriy Vyatkin, Folie 68). Laut Quelle sperrt ein `E_PERMIT` das Ereignis `TRIGGER` vor dem Baustein `TrueUntil`, **ohne `TrueUntil` selbst zu verändern**: Der Schritt läuft nur, solange die Bedingung `TE` den Wert `TRUE` hat.

Schnittstelle der SubApp

Name	Art	Typ	Beschreibung
`TRIGGER`	Ereignis-Eingang	`Event`	Schritt anfordern – läuft nur, solange `TE` = `TRUE` ist
`DONE`	Ereignis-Ausgang	`Event`	Der Schritt ist fertig (kommt nur, wenn `TE` beim Eintreffen von `TRIGGER` `TRUE` war)
`TE`	Daten-Eingang	`BOOL` (Start `TRUE`)	Decorator-Bedingung
`InPosition`	Daten-Eingang	`BOOL`	Demo: simuliert, dass der Aktor des Schritts seine Position erreicht hat

Verwendete Funktionsbausteine (FBs)

Instanz	Typ	Parameter	Erklärung
`Gate`	E_PERMIT	–	Lässt `EI` nur zum Ausgang `EO` durch, wenn `PERMIT` = `TRUE`

MeinWorker	DesignPatterns::ChainOfActionsPattern::TrueUntil		Der „dekorierte“ Schritt: fährt eine Position an und wartet, bis `inPosition` `TRUE` wird, dann meldet er `DONE` (siehe Chain-of-Actions-Demo)

Programmablauf und Verbindungen

1. `TRIGGER` ⟶ `Gate.EI`; `TE` ⟶ `Gate.PERMIT`.
2. `Gate.EO` ⟶ `Worker.TRIGGER`: Das Ereignis erreicht den Schritt nur bei `TE` = `TRUE`.
3. `InPosition` ⟶ `Worker.inPosition`.
4. `Worker.DONE` ⟶ `DONE`.

Mit `TE` = `FALSE` bleibt der Schritt unberührt: `TRIGGER` wird am `Gate` verworfen und es kommt kein `DONE`.

Zusammenfassung

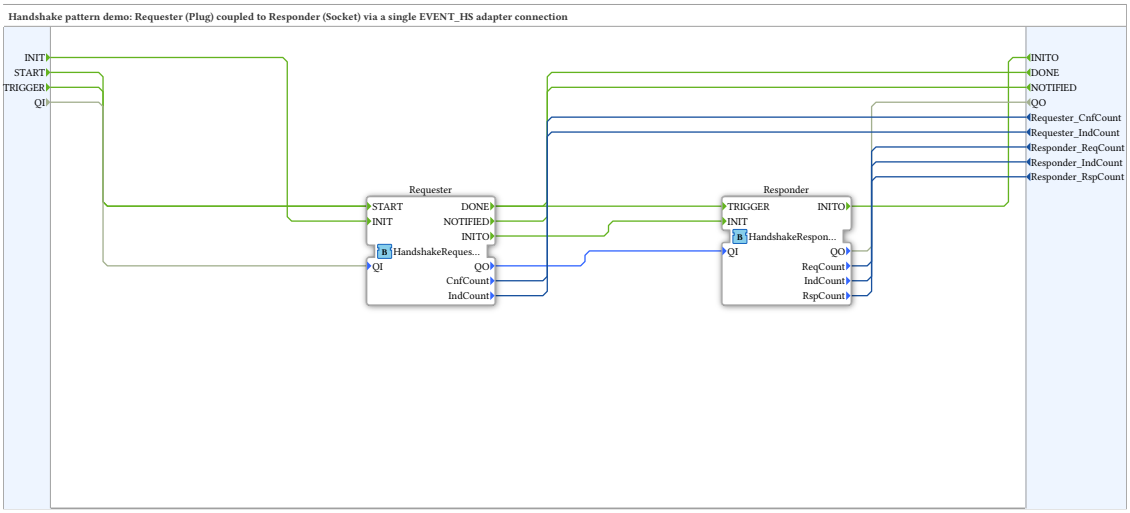
`DecoratorDemo` zeigt das Decorator-Muster in der einfachsten Form: ein vorgeschalteter `E_PERMIT` ergänzt eine Freigabebedingung, ohne den dekorierten Baustein `TrueUntil` zu ändern.

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

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

HandshakePattern

HandshakePatternDemo: Handshake-Muster (Requester/Responder)



• — ○ • — ○ • — ○ •

Einleitung

`HandshakePatternDemo` zeigt das Handshake-Muster: ein Requester (Plug) ist mit einem Responder (Socket) über **eine einzige** `EVENT_HS`-Adapterverbindung gekoppelt (Kommentar im Typ). Diese eine Verbindung ersetzt vier getrennte Ereignisverbindungen `REQ`/`CNF`/`IND`/`RSP`.

Schnittstelle der SubApp: Ereigniseingänge `INIT`, `START` (Requester sendet ein `REQ`) und `TRIGGER` (Responder sendet ein unaufgefordertes `IND`); Ereignisausgänge `INITO`, `DONE` (Requester hat das `CNF` erhalten) und `NOTIFIED` (Requester hat ein `IND` erhalten und mit `RSP` beantwortet); Eingang `QI`, Ausgang `QO` und die Zähler `Requester_CnfCount`, `Requester_IndCount`, `Responder_ReqCount`, `Responder_IndCount`, `Responder_RspCount` (alle `UINT`).

Verwendete Funktionsbausteine (FBs)

Instanz	Typ	Parameter	Erklärung

MeinRequester	DeineBibliothek::DeinPatterns::HandshakePattern::HandshakeRequester	Plug HS (adapter: ::types::bidirectional::Handshake::EVENT_HS)	sendet REQ/RSP, empfängt CNF/IND; zählt CnfCount und IndCount.
MeinResponder	DeineBibliothek::DeinPatterns::HandshakePattern::HandshakeResponder	Socket HS	empfängt REQ/RSP, sendet CNF/IND; zählt ReqCount, IndCount, RspCount.

Programmablauf und Verbindungen

Adapterverbindung: `Requester.HS` ⟶ `Responder.HS` (Requester = Plug ist Quelle, Responder = Socket ist Ziel).

Ereignisverbindungen:

1. `INIT` ⟶ `Requester.INIT`; `Requester.INITO` ⟶ `Responder.INIT`; `Responder.INITO` ⟶ `INITO` (Initialisierung läuft durch beide Bausteine).
2. `START` ⟶ `Requester.START`; `TRIGGER` ⟶ `Responder.TRIGGER`.
3. `Requester.DONE` ⟶ `DONE`; `Requester.NOTIFIED` ⟶ `NOTIFIED`.

Datenverbindungen: `QI` ⟶ `Requester.QI`; `Requester.QO` ⟶ `Responder.QI`; `Responder.QO` ⟶ `QO`; die fünf Zähler werden von den beiden Bausteinen nach außen geführt.

Beide Bausteine sind Basis-FBs im Projektordner (`HandshakeRequester.fbt`, `HandshakeResponder.fbt`).

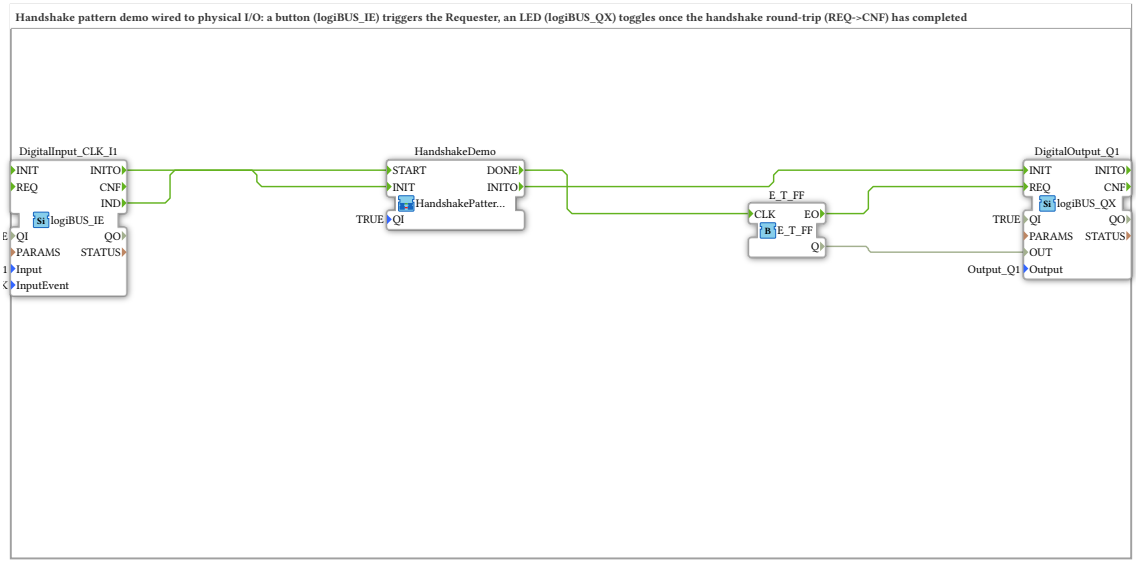
Zusammenfassung

Das Demo zeigt, dass der gesamte Anfrage-/Antwort-Dialog zwischen zwei Bausteinen über eine einzige Adapterverbindung läuft; Zähler machen die gesendeten und empfangenen Telegramme sichtbar.

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

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

HandshakePatternDemoIO: Handshake-Muster an echter I/O



• – ○ • – ○ • – ○ •

Einleitung

`HandshakePatternDemoIO` verdrahtet die Handshake-Demo `HandshakePatternDemo` (IEC-61499-Grundkurs, Modul 6, Valeriy Vyatkin) mit physischer I/O: Ein Taster (`logiBUS_IE`) löst den Requester aus, eine LED (`logiBUS_QX`) wechselt ihren Zustand, sobald der Handshake-Rundlauf (REQ ⟶ CNF) abgeschlossen ist. Die Taster-/LED-Verdrahtung ist laut Versionsvermerk dem Beispiel `test_B/Uebungen/Uebung_004a` nachempfunden. Die Schnittstelle der SubApp ist leer.

Verwendete Funktionsbausteine (FBs)

Instanz	Typ	Parameter	Erklärung
`DigitalInput_CLK_I1`	logiBUS_IE	`QI = TRUE, Input = Input_I1, InputEvent = BUTTON_SINGLE_CLICK`	Taster (Einzelklick)
`Me[illegible]`	`HandshakePatterns::HandshakePattern::HandshakePatternDemo` (SubApp)	`QI = TRUE`	Handshake-Demo mit Requester und Responder; Eingang `START`, Ausgang `DONE`
`E_T_FF`	E_T_FF	–	Toggle-Flipflop: wechselt bei jedem `CLK` den Zustand
`DigitalOutput_Q1`	logiBUS_QX	`QI = TRUE, Output = Output_Q1`	LED an Ausgang Q1

Programmablauf und Verbindungen

1. **Start**: `DigitalInput_CLK_I1.IND` ⟶ `HandshakeDemo.START` – der Tastendruck lässt den Requester sein `REQ` senden.
2. **Rückmeldung**: `HandshakeDemo.DONE` ⟶ `E_T_FF.CLK` – die LED wird erst umgeschaltet, wenn das `CNF` des Responders tatsächlich zurückgekommen ist.
3. **LED**: `E_T_FF.EO` ⟶ `DigitalOutput_Q1.REQ`, `E_T_FF.Q` ⟶ `DigitalOutput_Q1.OUT`.
4. **Initialisierung**: `DigitalInput_CLK_I1.INITO` ⟶ `HandshakeDemo.INIT`, danach `HandshakeDemo.INITO` ⟶ `DigitalOutput_Q1.INIT`.

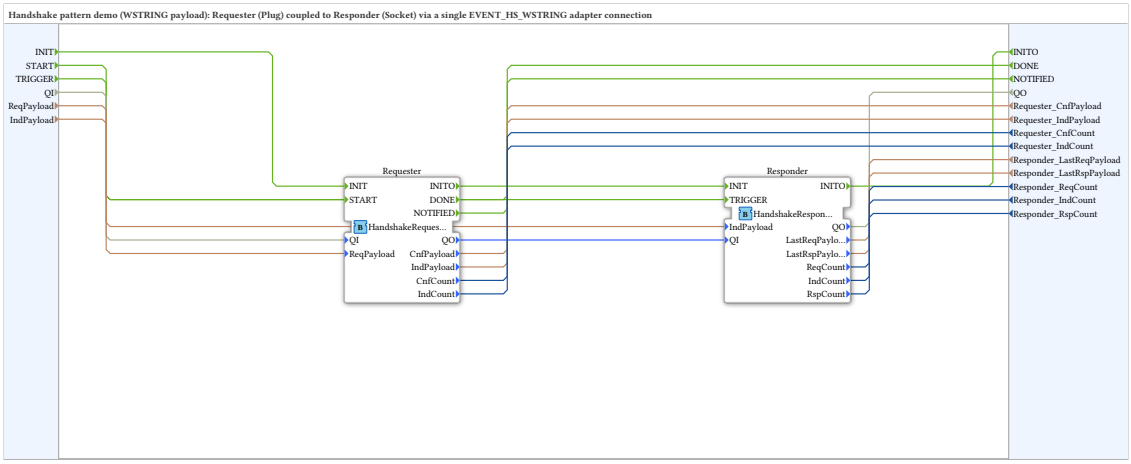
Zusammenfassung

`HandshakePatternDemoIO` macht den Handshake sichtbar: Jeder Tastendruck schaltet die LED erst nach abgeschlossenem Anfrage-/Bestätigungs-Rundlauf um.

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

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

HandshakePatternDemoWSTRING: Handshake-Muster mit WSTRING-Nutzdaten



• – ○ • – ○ • – ○ •

Einleitung

`HandshakePatternDemoWSTRING` ist die Variante von `HandshakePatternDemo` mit Nutzdaten: Requester (Plug) und Responder (Socket) sind über **eine einzige** `EVENT_HS_WSTRING`-Adapterverbindung gekoppelt, die `REQ`/`CNF`/`IND`/`RSP` jeweils samt `WSTRING`-Nutzdaten transportiert (Kommentar im Typ).

Zusätzlich zur Schnittstelle des Basis-Demos (`INIT`, `START`, `TRIGGER`, `INITO`, `DONE`, `NOTIFIED`, `QI`, `QO`, fünf `UINT`-Zähler) gibt es die Eingänge `ReqPayload` (Vorgabe `"push,100"`) und `IndPayload` (Vorgabe `"status,ok"`) sowie die Ausgänge `Requester_CnfPayload`, `Requester_IndPayload`, `Responder_LastReqPayload` und `Responder_LastRspPayload` (alle `WSTRING`).

Verwendete Funktionsbausteine (FBs)

Instanz	Typ	Parameter	Erklärung
`Meins::RequesterPatterns::HandshakePattern::HandshakeRequesterWSTRING`			Plug `HS` (`EVENT_HS_WSTRING`): sendet `REQ` mit `ReqPayload`, empfängt `CNF`/`IND` mit Nutzdaten und antwortet auf `IND` mit `RSP`.
`Meins::ResponderPatterns::HandshakePattern::HandshakeResponderWSTRING`			Socket `HS`: empfängt `REQ`/`RSP`, sendet `CNF` und (bei `TRIGGER`) `IND` mit `IndPayload`.

Programmablauf und Verbindungen

Adapterverbindung: `Requester.HS` ⟶ `Responder.HS` (Requester = Plug ist Quelle, Responder = Socket ist Ziel).

Ereignisverbindungen: `INIT` ⟶ `Requester.INIT` ⟶ `Requester.INITO` ⟶ `Responder.INIT` ⟶ `Responder.INITO` ⟶ `INITO`; `START` ⟶ `Requester.START`; `TRIGGER` ⟶ `Responder.TRIGGER`; `Requester.DONE` ⟶ `DONE`; `Requester.NOTIFIED` ⟶ `NOTIFIED`.

Datenverbindungen:

1. `QI` ⟶ `Requester.QI`; `Requester.QO` ⟶ `Responder.QI`; `Responder.QO` ⟶ `QO`.
2. `ReqPayload` ⟶ `Requester.ReqPayload`; `IndPayload` ⟶ `Responder.IndPayload`.
3. `Requester.CnfPayload`, `Requester.IndPayload`, `Requester.CnfCount`, `Requester.IndCount` ⟶ gleichnamige Ausgänge `Requester_*`.
4. `Responder.LastReqPayload`, `Responder.LastRspPayload`, `Responder.ReqCount`, `Responder.IndCount`, `Responder.RspCount` ⟶ Ausgänge `Responder_*`.

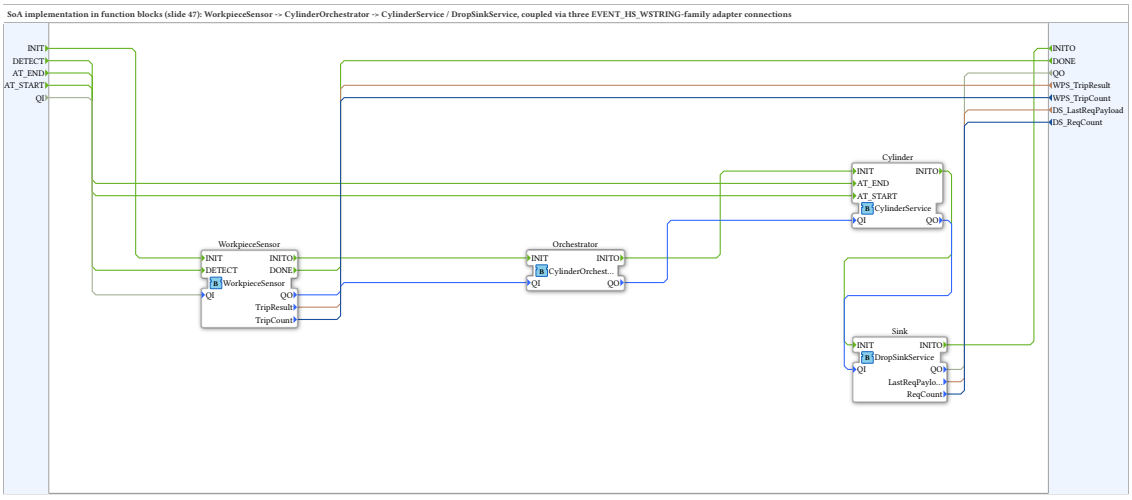
Zusammenfassung

Wie das Basis-Demo, jedoch transportiert dieselbe eine Adapterverbindung zusätzlich Text-Nutzdaten (`WSTRING`) in beide Richtungen.

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

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

MessageExchangeDemo: Nachrichtenaustausch über Handshake-Adapter



• – ○ • – ○ • – ○ •

Einleitung

`MessageExchangeDemo` setzt die „SoA-Implementierung in Funktionsbausteinen“ (IEC-61499-Grundkurs, Modul 6, Valeriy Vyatkin, Folie 47) um und rekonstruiert den CylH-Nachrichtenaustausch (Folien 42–46) mit **vier zusammenarbeitenden Funktionsbausteinen**: `WorkpieceSensor` → `CylinderOrchestrator` → `CylinderService` / `DropSinkService`. Gekoppelt werden sie über drei Adapter-Verbindungen aus der Familie `EVENT_HS_WSTRING`.

Schnittstelle der SubApp

Name	Art	Typ	Beschreibung
`INIT` / `INITO`	Ereignis	`EInit`	Initialisierung / Bestätigung
`DETECT`	Ereignis-Eingang	`Event`	Simulierter Sensor WPS: Werkstück angekommen, startet den Rundlauf
`AT_END`	Ereignis-Eingang	`Event`	Simulierter Sensor CYL.end: Stößel in Endlage
`AT_START`	Ereignis-Eingang	`Event`	Simulierter Sensor CYL.start: Stößel wieder in Startlage
`DONE`	Ereignis-Ausgang	`Event`	`WorkpieceSensor` hat das `CNF` erhalten: Rundlauf abgeschlossen
`QI` / `QO`	Daten	`BOOL`	Ereignis-Qualifizierer
`WPS_TripResult`	Daten-Ausgang	`WSTRING`	Nutzlast des letzten `CNF` am `WorkpieceSensor` (erwartet `RSP_CYL_TRIP_COMPLETE`)
`WPS_TripCount`	Daten-Ausgang	`UINT`	Anzahl abgeschlossener Rundläufe
`DS_LastReqPayload`	Daten-Ausgang	`WSTRING`	Nutzlast des letzten `REQ` am `DropSinkService` (erwartet `REQ_DROP`)
`DS_ReqCount`	Daten-Ausgang	`UINT`	Anzahl empfangener Drop-`REQ`

Verwendete Funktionsbausteine (FBs)

Alle vier Bausteine liegen im Paket `Meins::DesingPatterns::HandshakePattern`.

Instanz	Typ	Erklärung
`WorkpieceSensor`	`WorkpieceSensor`	Startet bei `DETECT` die Anfrage `REQ_TRIP` über den Adapter `ACTIVATE`; liefert `TripResult`, `TripCount` und `DONE`
`Orchestrator`	`CylinderOrchestrator`	Nimmt die Anfrage vom Sensor entgegen (`SRSP`), leitet sie an den Zylinder weiter (`SREQ1`) und fordert das Ablegen beim Sink an (`SREQ2`)
`Cylinder`	`CylinderService`	Zylinder-Dienst mit Ereignissen `AT_END` und `AT_START`; antwortet über `SRSP`
`Sink`	`DropSinkService`	Senken-Dienst; liefert `LastReqPayload` und `ReqCount`

Programmablauf und Verbindungen

- **Adapter-Kopplung** (jeweils mit Kommentar in der Quelle):
  - `WorkpieceSensor.ACTIVATE` → `Orchestrator.SRSP`: Der Sensor fordert den Rundlauf an (`REQ_TRIP`); der Orchestrator antwortet, sobald der gesamte Rundlauf als abgeschlossen bestätigt ist.
  - `Orchestrator.SREQ1` → `Cylinder.SRSP`: Der Orchestrator reicht die Anfrage weiter und reagiert auf das `IND` mitten im Rundlauf (Endlage erreicht) und das abschließende `CNF` (Rundlauf fertig).
  - `Orchestrator.SREQ2` → `Sink.SRSP`: Der Orchestrator fordert das Ablegen an, sobald der Zylinder die Endlage meldet.
- **Initialisierungskette**: `INIT` → `WorkpieceSensor` → `Orchestrator` → `Cylinder` → `Sink` → `INITO`; entsprechend wird `QI` durch die vier Bausteine zu `QO` durchgereicht.

- **Ereignisse**: `DETECT` ⟶ `WorkpieceSensor.DETECT`, `AT_END` ⟶ `Cylinder.AT_END`, `AT_START` ⟶ `Cylinder.AT_START`, `WorkpieceSensor.DONE` ⟶ `DONE`.
- **Daten**: `WorkpieceSensor.TripResult`/`TripCount` und `Sink.LastReqPayload`/`ReqCount` werden nach außen geführt.

Zusammenfassung

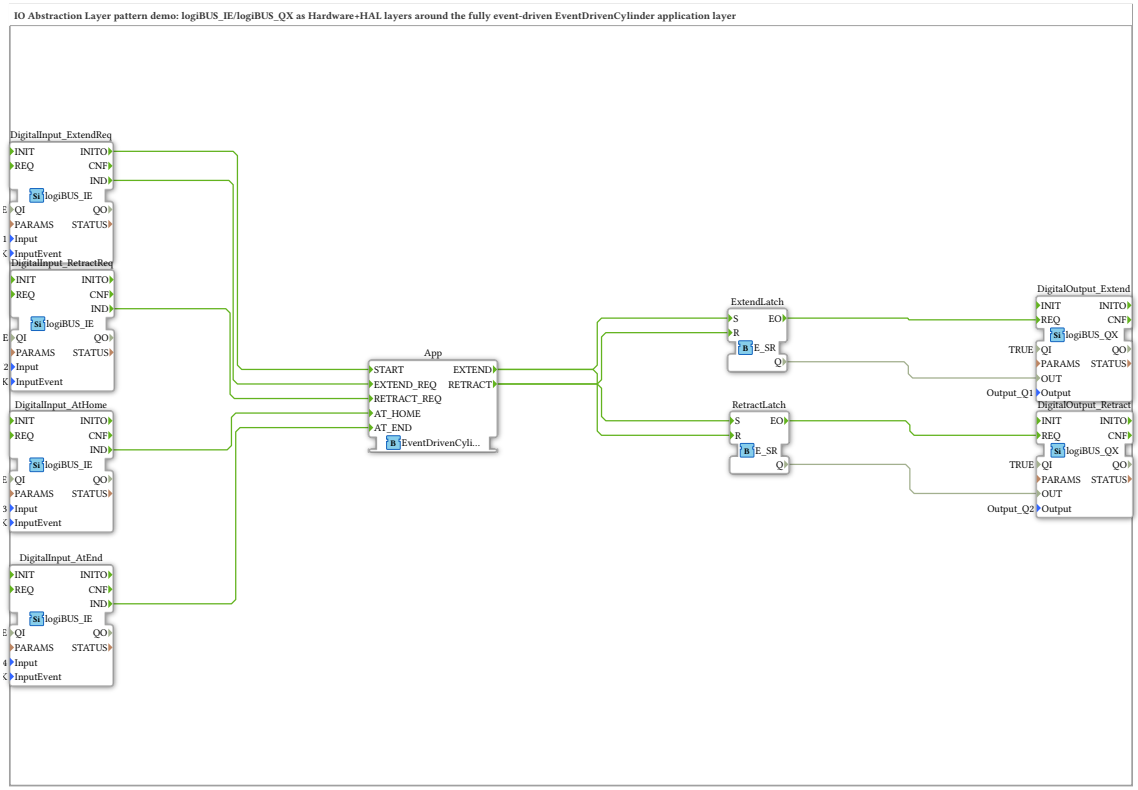
`MessageExchangeDemo` zeigt den Handshake-Gedanken als Nachrichtenaustausch zwischen vier Diensten: Sensor, Orchestrator, Zylinder und Senke tauschen Anfragen und Bestätigungen ausschließlich über Adapter aus; die Zähler und Nutzlasten an der Schnittstelle machen den Ablauf beobachtbar.

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

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

IOAbstractionPattern

IOAbstractionDemo: IO-Abstraktionsschicht-Muster



• – ○ • – ○ • – ○ •

Einleitung

`IOAbstractionDemo` zeigt das IO-Abstraktionsschicht-Muster: `logiBUS_IE` und `logiBUS_QX` bilden die Hardware- bzw. HAL-Schicht um die vollständig ereignisgesteuerte Anwendungsschicht `EventDrivenCylinder` (Kommentar im Typ). Der Zylinder wird über vier Eingänge (Ausfahren-/Einfahren-Anforderung, Endlagen Home/Ende) angesteuert und schaltet zwei Ausgänge (Ausfahren, Einfahren). Die SubApp hat keine eigene Schnittstelle.

Verwendete Funktionsbausteine (FBs)

Instanz	Typ	Parameter	Erklärung
`DigitalInput_ExtendReq`	`logiBUS::io::DI::logiBUS_IE`	`Input = Input_I1, InputEvent =`	Anforderung Ausfahren.

		BUTTON_SINGLE_CLICK, QI = TRUE	
DigitalInput_RetractReq	logiBUS::io::DI::logiBUS_DI	Input = Input_I2, InputEvent = BUTTON_SINGLE_CLICK, QI = TRUE	Anforderung Einfahren.
DigitalInput_AtHome	logiBUS::io::DI::logiBUS_DI	Input = Input_I3, InputEvent = BUTTON_SINGLE_CLICK, QI = TRUE	Meldung Endlage Home.
DigitalInput_AtEnd	logiBUS::io::DI::logiBUS_DI	Input = Input_I4, InputEvent = BUTTON_SINGLE_CLICK, QI = TRUE	Meldung Endlage Ende.
Meins::DesingPatterns::PurelyEventDrivenPattern::EventDrivenCylinder			Ereignisgesteuerter Zylinder ohne BOOL-Datenpins (Zustände HOME, EXTENDING, EXTENDED, RETRACTING usw.).
ExtendLatch	iec61499::events::E_SR	-	Speicher für die Ausfahren-Ansteuerung.
RetractLatch	iec61499::events::E_SR	-	Speicher für die Einfahren-Ansteuerung.
DigitalOutput_Extend	logiBUS::io::DQ::logiBUS_QX	Output = Output_Q1, QI = TRUE	Ausgang Ausfahren.
DigitalOutput_Retract	logiBUS::io::DQ::logiBUS_QX	Output = Output_Q2, QI = TRUE	Ausgang Einfahren.

Programmablauf und Verbindungen

Eingangsseite (Hardware → Anwendung):

1. `DigitalInput_ExtendReq.INITO` → `App.START` (Kommentar: „Bootstrap: fires once at deployment“).
2. `DigitalInput_ExtendReq.IND` → `App.EXTEND_REQ`; `DigitalInput_RetractReq.IND` → `App.RETRACT_REQ`.
3. `DigitalInput_AtHome.IND` → `App.AT_HOME`; `DigitalInput_AtEnd.IND` → `App.AT_END`.

Ausgangsseite (Anwendung → Hardware):

1. `App.EXTEND` → `ExtendLatch.S` und `RetractLatch.R`.
2. `App.RETRACT` → `RetractLatch.S` und `ExtendLatch.R`.
3. `ExtendLatch.EO` → `DigitalOutput_Extend.REQ`, `ExtendLatch.Q` → `DigitalOutput_Extend.OUT`.
4. `RetractLatch.EO` → `DigitalOutput_Retract.REQ`, `RetractLatch.Q` → `DigitalOutput_Retract.OUT`.

Die Anwendung kennt nur Ereignisse; die Umsetzung auf Pegel übernehmen die beiden E_SR-Speicher und die `logiBUS_QX`-Ausgänge. `EventDrivenCylinder` ist ein Basis-FB im Projektordner `DesingPatterns/PurelyEventDrivenPattern`.

Zusammenfassung

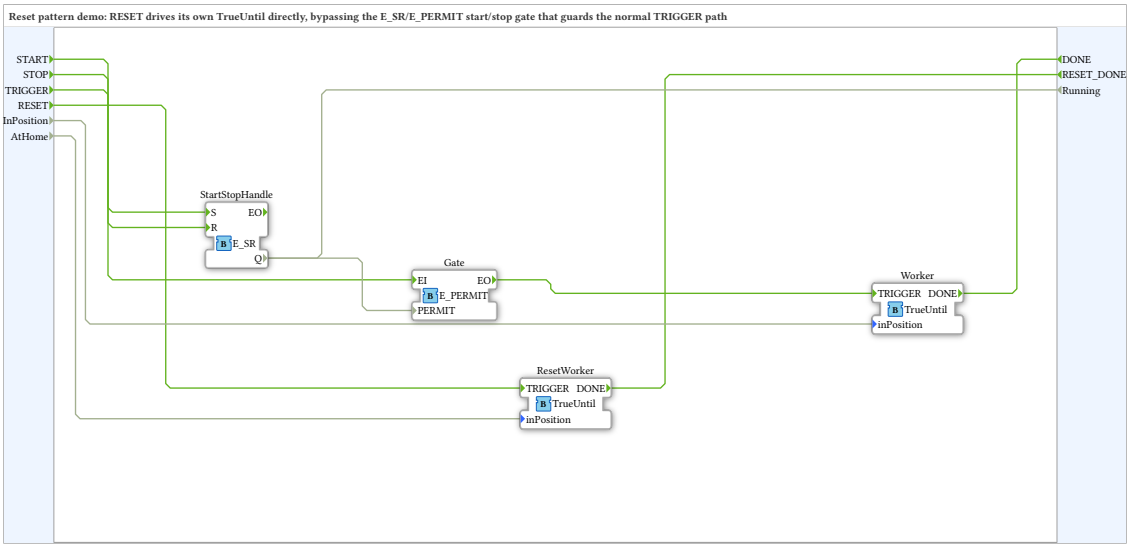
Das Demo trennt Hardwarezugriff (logiBUS-Ein-/Ausgänge, Speicher) von der Anwendungslogik (`EventDrivenCylinder`), die ausschließlich mit Ereignissen arbeitet.

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

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

ResetPattern

ResetDemo: Reset-Muster (Umgehung des Start/Stopp-Tors)



• – ○ • – ○ • – ○ •

Einleitung

`ResetDemo` ist die Demo zum **Reset-Muster** (IEC-61499-Grundkurs, Modul 6, Valeriy Vyatkin, Folie 71): `RESET` treibt einen eigenen `TrueUntil` direkt an und **umgeht dabei das Start/Stopp-Tor** (`E_SR` + `E_PERMIT`), das den normalen `TRIGGER`-Pfad schützt. Der Reset funktioniert daher auch im gestoppten Zustand.

Schnittstelle der SubApp

Name	Art	Typ	Beschreibung
`START`	Ereignis-Eingang	`Event`	Anlage starten: öffnet das Tor für den normalen `TRIGGER`-Pfad
`STOP`	Ereignis-Eingang	`Event`	Anlage stoppen: schließt das Tor
`TRIGGER`	Ereignis-Eingang	`Event`	Normalen Schritt anfordern – erreicht ihn nur bei gestarteter Anlage
`RESET`	Ereignis-Eingang	`Event`	Reset-Schritt anfordern – umgeht das Tor, funktioniert auch im Stopp
`DONE`	Ereignis-Ausgang	`Event`	Der normale Schritt ist fertig
`RESET_DONE`	Ereignis-Ausgang	`Event`	Der Reset-Schritt ist fertig
`InPosition`	Daten-Eingang	`BOOL`	Demo: Aktor des normalen Schritts hat die Position erreicht
`AtHome`	Daten-Eingang	`BOOL`	Demo: Aktor des Reset-Schritts hat die Grundstellung erreicht
`Running`	Daten-Ausgang	`BOOL`	Aktueller Start/Stopp-Zustand (`E_SR.Q`); `RESET` ignoriert ihn

Verwendete Funktionsbausteine (FBs)

Instanz	Typ	Parameter	Erklärung

StartStopHandle	E_SR	–	Start/Stopp-Speicher: START setzt (S), STOP setzt zurück (R)
Gate	E_PERMIT	–	Tor für den normalen Pfad, Freigabe aus StartStopHandle.Q
Worker	Basic.DesignPatterns::ChainOfActionsPattern::TrueUntil		Normaler Schritt
ResetWorker	Basic.DesignPatterns::ChainOfActionsPattern::TrueUntil		Reset-Schritt (eigene Instanz)

Programmablauf und Verbindungen

- **Normaler Pfad**: TRIGGER → Gate.EI → Gate.EO → Worker.TRIGGER; Worker.DONE → DONE; InPosition → Worker.inPosition. Das Tor ist offen, solange StartStopHandle.Q = TRUE ist (StartStopHandle.Q → Gate.PERMIT).
- **Start/Stopp**: START → StartStopHandle.S, STOP → StartStopHandle.R; der Zustand wird zusätzlich als Running ausgegeben.
- **Reset-Pfad**: RESET → ResetWorker.TRIGGER (laut Kommentar „bypasses StartStopHandle/Gate entirely – works even while stopped“); ResetWorker.DONE → RESET_DONE; AtHome → ResetWorker.inPosition.

Zusammenfassung

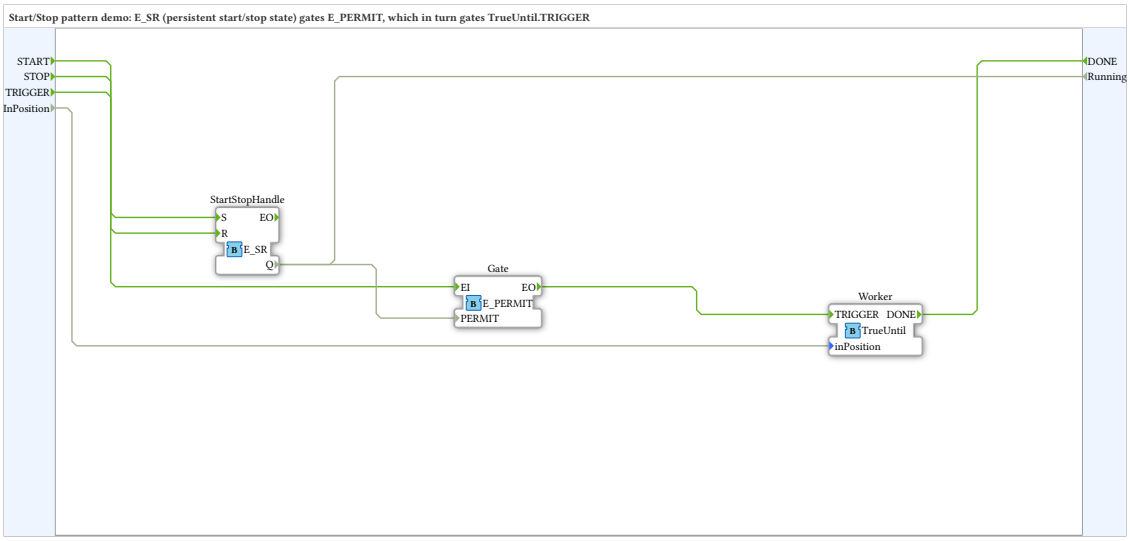
ResetDemo zeigt, wie ein Reset-Befehl über einen eigenen, vom Start/Stopp-Tor unabhängigen Pfad geführt wird, damit er auch bei gestoppter Anlage wirkt.

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

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

StartStopPattern

StartStopDemo: Start/Stop-Muster



• – ○ • – ○ • – ○ •

Einleitung

StartStopDemo zeigt das Start/Stop-Muster: ein E_SR hält den Start/Stop-Zustand dauerhaft und gibt über E_PERMIT die Weitergabe des Ereignisses an TrueUntil.TRIGGER frei oder sperrt sie (Kommentar im Typ).

Schnittstelle: Ereigniseingänge START, STOP und TRIGGER; Ereignisausgang DONE; Eingang InPosition (BOOL, simuliert die Rückmeldung der Aktorik); Ausgang Running (BOOL, TRUE, solange gestartet).

Verwendete Funktionsbausteine (FBs)

Instanz	Typ	Parameter	Erklärung
`StartStopHandle`	`iec61499::events::E_SR`	-	Set/Reset-Speicher: `START` ⟶ `S`, `STOP` ⟶ `R`; Ausgang `Q` ist der Start/Stop-Zustand.
`Gate`	`iec61499::events::E_PERMIT`	-	Gibt das Ereignis `EI` nur als `EO` weiter, solange `PERMIT = TRUE` ist.
`Worker`	`DesingPatterns::ChainOfActionsPattern::TrueUntil`	-	Schritt-Baustein aus dem Chain-of-Actions-Muster.

Programmablauf und Verbindungen

Ereignisverbindungen:

1. `START` ⟶ `StartStopHandle.S`; `STOP` ⟶ `StartStopHandle.R`.
2. `TRIGGER` ⟶ `Gate.EI`; `Gate.EO` ⟶ `Worker.TRIGGER`.
3. `Worker.DONE` ⟶ `DONE`.

Datenverbindungen:

1. `StartStopHandle.Q` ⟶ `Gate.PERMIT`.
2. `StartStopHandle.Q` ⟶ `Running`.
3. `InPosition` ⟶ `Worker.inPosition`.

Ein `TRIGGER` erreicht `Worker` nur, wenn die Anlage zuvor gestartet wurde (laut Schnittstellenkommentar feuert `DONE` nur dann). `TrueUntil` ist ein Basis-FB im Projektordner `DesingPatterns/ChainOfActionsPattern`.

Zusammenfassung

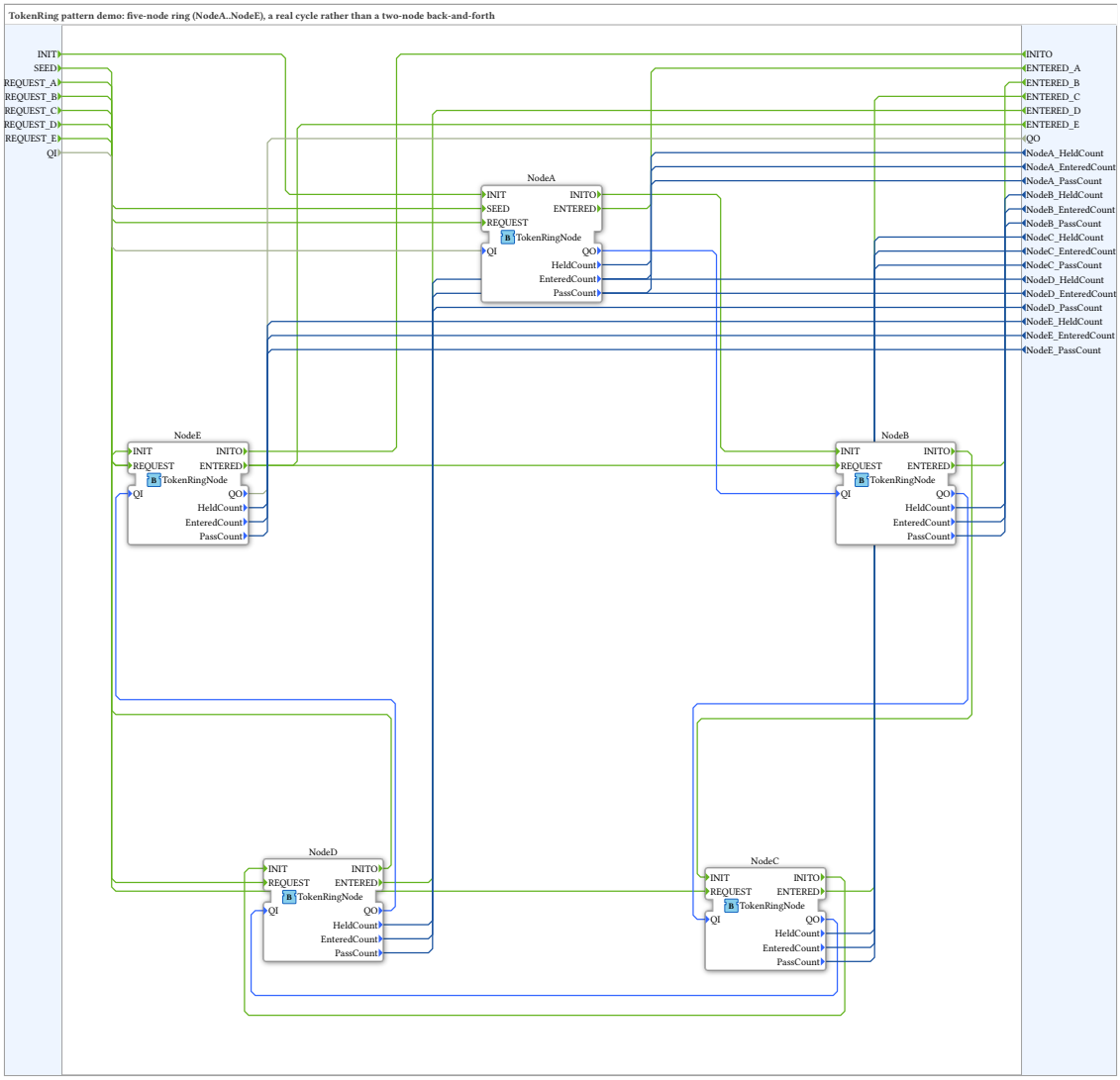
Das Demo trennt den dauerhaften Start/Stop-Zustand (E_SR) von der Freigabe einzelner Aufträge (E_PERMIT) und lässt den Arbeitsbaustein unverändert.

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

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

TokenRingPattern

TokenRingPatternDemo: Token-Ring-Muster mit fünf Knoten



• — ○ • — ○ • — ○ •

Einleitung

`TokenRingPatternDemo` ist die Demo zum **Mutual-Exclusion-/Token-Ring-Muster** (IEC-61499-Grundkurs, Modul 6, Valeriy Vyatkin). Laut Quelle wurde sie in Version 1.1 von einem Knotenpaar auf einen **echten Ring mit fünf Knoten** (`NodeA`…`NodeE`) erweitert, weil zwei Knoten nur ein Hin-und-Her, aber keinen Ring ergeben.

Schnittstelle der SubApp

Name	Art	Typ	Beschreibung
`INIT` / `INITO`	Ereignis	`EInit`	Initialisierung / Bestätigung
`SEED`	Ereignis-Eingang	`Event`	Ring starten: Token einmalig bei `NodeA` einspeisen (nach `INIT`)
`REQUEST_A` … `REQUEST_E`	Ereignis-Eingänge	`Event`	Knoten fordert den kritischen Abschnitt für sein nächstes Token-Halten an

ENTERED_A … ENTERED_E	Ereignis-Ausgänge	Event	Knoten hat seinen kritischen Abschnitt betreten
QI / QO	Daten	BOOL	Ereignis-Qualifizierer
NodeX_HeldCount	Daten-Ausgang	UINT	Wie oft Knoten X das Token erhalten hat
NodeX_EnteredCount	Daten-Ausgang	UINT	Wie oft Knoten X den kritischen Abschnitt genutzt hat
NodeX_PassCount	Daten-Ausgang	UINT	Anzahl bestätigter Weitergaben von Knoten X

(X = A…E)

Verwendete Funktionsbausteine (FBs)

Instanz	Typ	Erklärung
NodeA…NodeE	Meins::DesingPatterns::TokenRingPattern::TokenRingNode	Knoten des Rings. Hat die Adapter MTXIN (Token vom Vorgänger empfangen, Empfang bestätigen) und MTXOUT (Token an den Nachfolger geben, Bestätigung abwarten) vom Typ adapter::types::bidirectional::TokenRing::TokenRing; Ereignisse: REQUEST, SEED, ENTERED, EXITED; Zähler HeldCount, EnteredCount, PassCount

Innerhalb von TokenRingNode merkt REQUEST einen Wunsch (Want); beim Eintreffen des Tokens betritt der Knoten den kritischen Abschnitt nur, wenn Want gesetzt ist, und reicht das Token danach an den Nachfolger weiter. SEED schiebt das Token direkt in die Weitergabe.

Programmablauf und Verbindungen

- **Token-Weg** (Adapter-Verbindungen MTXOUT ⟶ MTXIN): NodeA ⟶ NodeB ⟶ NodeC ⟶ NodeD ⟶ NodeE ⟶ zurück zu NodeA („closing the ring“).
- **Initialisierung**: INIT ⟶ NodeA.INIT, dann die INITO⟶INIT-Kette über NodeB, NodeC, NodeD bis NodeE.INITO ⟶ INITO; QI wird entsprechend von Knoten zu Knoten bis QO durchgereicht.
- **Bedienung**: SEED ⟶ NodeA.SEED; REQUEST_x ⟶ Node<x>.REQUEST; Node<x>.ENTERED ⟶ ENTERED_x.
- **Beobachtung**: Die Zähler HeldCount, EnteredCount und PassCount jedes Knotens sind als Ausgänge der SubApp herausgeführt.

Zusammenfassung

TokenRingPatternDemo zeigt gegenseitigen Ausschluss mit einem umlaufenden Token: Nur der Knoten, der das Token hält, darf den kritischen Abschnitt betreten; wer ihn nutzen will, merkt sich seinen Wunsch und kommt beim nächsten Token-Umlauf zum Zug.

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

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

2.1.1.8 GcfFunctionCall

GcfFunctionCall: Function in ST aus einer GCF aufrufen

• – ◦ • – ◦ • – ◦ •

Einleitung

Die Übung im Paket Meins::GcfFunctionCall zeigt, wie eine in Structured Text geschriebene Function (.fct) aus einer GlobalConstants-Datei (.gcf) heraus benutzt werden kann. Sie ist die Grundlage für eine spätere eigene Bibliothek (analog MyLib_AX_Option).

Inhalt

Datei	Zweck
`F_Double.fct`	ST-Function, `F_Double(Value) = Value * 2` (INT)
`const/FuncConst.gcf`	`BASE_VALUE := 7`, `DOUBLE_VALUE := F_Double(BASE_VALUE)`
`GcfFunctionCallTest`	Basic FB: liest `DOUBLE_VALUE` und ruft `F_Double` direkt auf, `Passed` ist TRUE, wenn beide 14 ergeben
`Uebung_011d_GcfFunc_AX`	VT-Anzeige: `DOUBLE_VALUE` als `rPhys` an `Q_NumericValue_PHYS` (`OutputNumber_N3_N`)

Programmablauf und Verbindungen

1. `FuncConst.gcf` importiert nichts. Die Function wird per vollem Namen `Meins::GcfFunctionCall::F_Double(...)` im `InitialValue` aufgerufen.
2. `Uebung_011d_GcfFunc_AX` importiert `DOUBLE_VALUE` und `OutputNumber_N3_N` (aus `Uebungen::const::UT::DefaultPool_Numeric`) und setzt sie als Parameter `rPhys` bzw. `stObj` des Bausteins `Q_NumericValue`.
3. Erwartung: Auf N3 erscheint 14 (Offset und Scale des Objekts wirken auf den Wert).

Ergebnis

Der Export akzeptiert den Funktionsaufruf als `InitialValue` und wertet ihn zur Exportzeit aus. Im Bootfile `test_AX_FORTE_PC_AX.fboot` steht `rPhys` direkt als `14.0` (= `F_Double(7)`), weder `F_Double` noch `DOUBLE_VALUE` tauchen dort auf. Die Function läuft also nicht zur Laufzeit auf dem Gerät. Voraussetzung: Die Argumente sind Konstanten.

Nicht geprüft: die Anzeige 14 auf N3 im VT (Offset und Scale des Objekts).

Hinweise

- `Comment=` in der GCF kurz halten (IDE-Wrap-Bug beim Speichern).
- `Uebung_011d.SUB` hieß wie `Uebungen/Uebung_011d_AX` und `_AUDI` (anderes Paket, gleiche Basis). Zur Vermeidung von Verwechslungen heißt sie jetzt `Uebung_011d_GcfFunc_AX`.

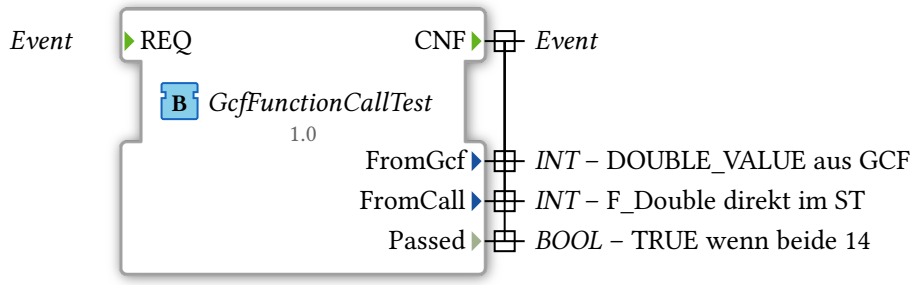
Zusammenfassung

Funktionsaufrufe mit konstanten Argumenten lassen sich in einer GCF als Initialwert nutzen. Der Wert wird beim Export berechnet und landet als Konstante im Bootfile.

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

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

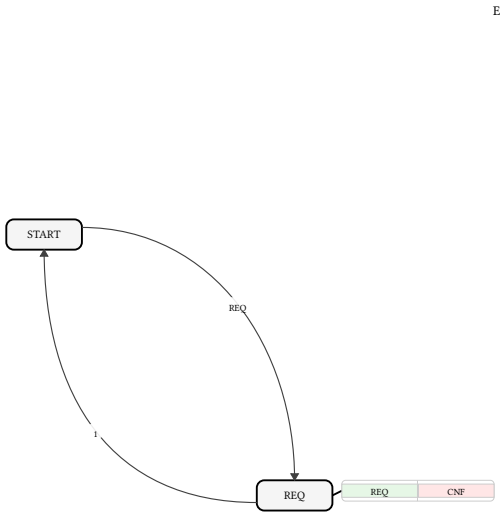
GcfFunctionCallTest: Test für die Function aus der GCF



• – ◦ • – ◦ • – ◦ •

Einleitung

`GcfFunctionCallTest` ist ein Basic FB im Paket `Meins::GcfFunctionCall`. Er vergleicht den Wert `DOUBLE_VALUE` aus der GCF mit einem direkten Aufruf der ST-Function `F_Double`.



Schnittstelle

Name	Typ	Erklärung
`REQ`	Ereigniseingang	Startet die Berechnung
`CNF`	Ereignisausgang (`With = FromGcf, FromCall, Passed`)	Bestätigung
`FromGcf`	`INT`	`DOUBLE_VALUE` aus der GCF
`FromCall`	`INT`	`F_Double(BASE_VALUE)`, direkt im ST berechnet
`Passed`	`BOOL`	TRUE, wenn beide Werte 14 sind

Programmablauf

Der ECC hat zwei Zustände: `START` und `REQ`. Bei `REQ` läuft der Algorithmus `REQ` und sendet `CNF`:

```
FromGcf := DOUBLE_VALUE;
FromCall := Meins::GcfFunctionCall::F_Double(BASE_VALUE);
Passed := (FromGcf = 14) AND (FromCall = 14);
```

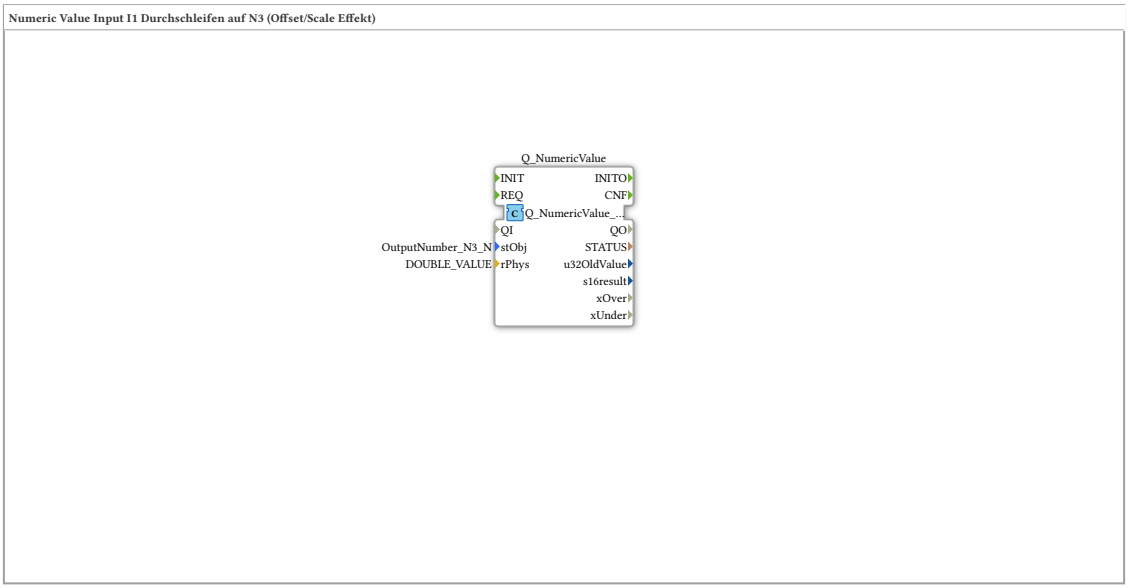
Zusammenfassung

Der Baustein prüft, dass der Wert aus der GCF und der direkte Funktionsaufruf zum selben Ergebnis führen (14).

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

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

Uebung_011d_GcfFunc_AX: Wert aus einer GCF-Function im VT anzeigen



• – ○ • – ○ • – ○ •

Einleitung

Die SubApp `Uebung_011d_GcfFunc_AX` im Paket `Meins::GcfFunctionCall` zeigt den Wert `DOUBLE_VALUE` aus einer GlobalConstants-Datei auf dem Universal Terminal an. `DOUBLE_VALUE` wird in der GCF mit der ST-Function `F_Double` berechnet (siehe GcfFunctionCall).

Verwendete Funktionsbausteine (FBs)

- **Q_NumericValue**: `isobus::UT::Q::Q_NumericValue_PHYS`
  - **Parameter**: `stObj = OutputNumber_N3_N`, `rPhys = DOUBLE_VALUE`
  - **Erklärung**: Gibt den physikalischen Wert an das Zahlenfeld N3 des Universal Terminals aus. Offset und Scale des Objekts wirken auf den Wert.

Programmablauf und Verbindungen

1. Der Baustein hat keine Verbindungen. Beide Werte kommen als importierte Konstanten (`DOUBLE_VALUE` aus `Meins::GcfFunctionCall::const::FuncConst`, `OutputNumber_N3_N` aus `Uebungen::const::UT::DefaultPool_Numeric`) direkt als Parameter.
2. Erwartung: Auf N3 erscheint 14 (`F_Double(7)`).

Zusammenfassung

Eine einzelne Konstante aus einer GCF reicht, um einen berechneten Wert im VT anzuzeigen. Die Berechnung `F_Double(7)` findet beim Export statt.

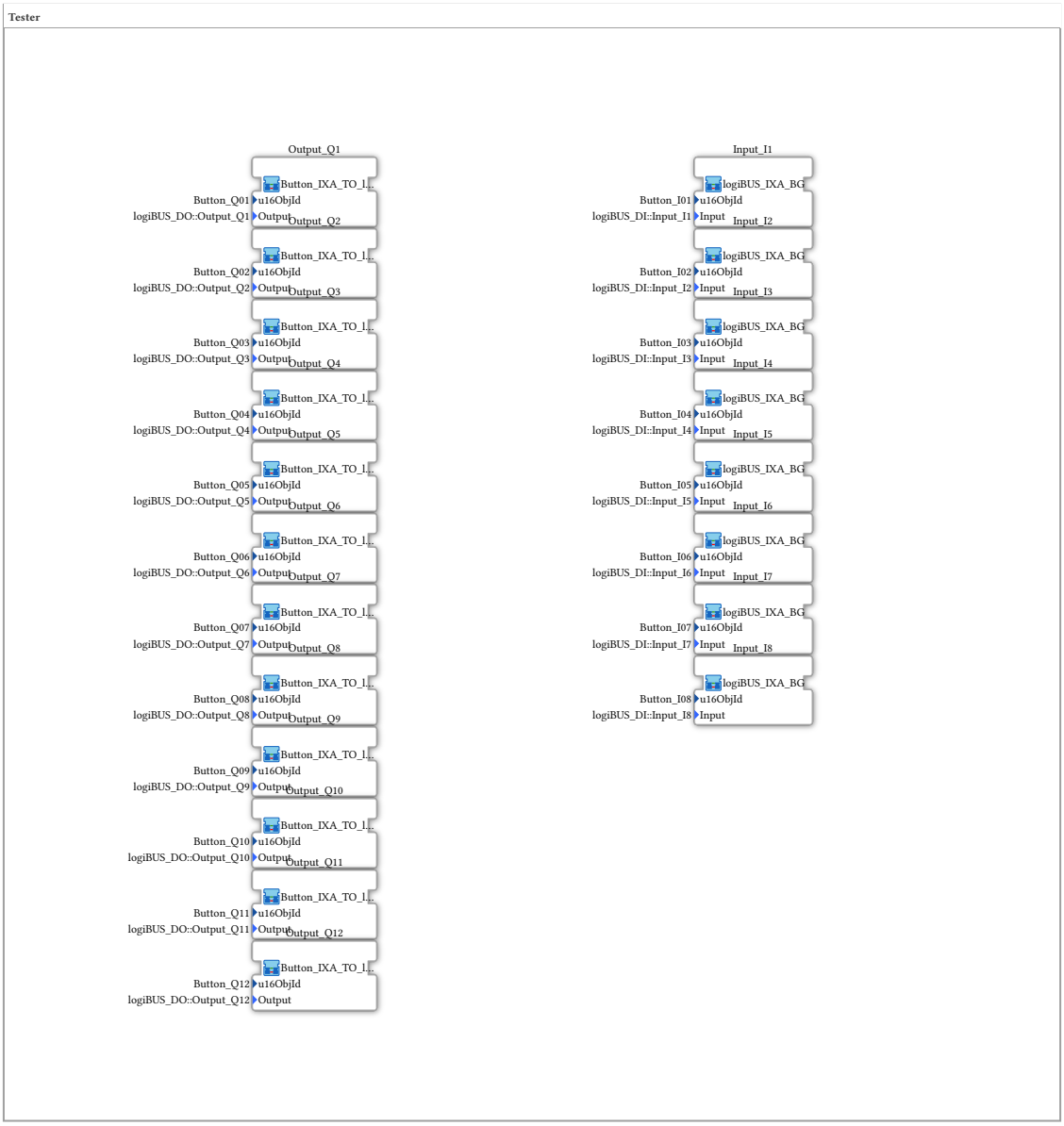
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

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

2.1.1.9 InputOutputTester

Button

InputOutputTesterBt: DIDO Tester (VT, ohne OPC-UA)



• — ○ • — ○ • — ○ •

Einleitung

`InputOutputTesterBt` ist die ursprüngliche, rein VT-basierte Fassung des DIDO-Testers für **8 digitale Eingänge und 12 digitale Ausgänge** — historisch die erste Version dieses Trainingsbeispiels, bevor die OPC-UA-Anbindung hinzukam. Der spätere `InputOutputTesterButton_DIDO_OPC_UA` übernimmt exakt dieselbe 8+12-Struktur, ersetzt aber `logiBUS_IXA_BG`/`Button_IXA_TO_logiBUS_QXA_BG` durch die `_OPC`-Varianten und ergänzt einen `SystemTickSender`.

Die Übung ist wie ihr OPC-UA-Nachfolger ein reines Top-Level-Composite ohne eigene Verbindungen — die gesamte Logik steckt in den wiederverwendbaren Sub-Bausteinen.

Verwendete Funktionsbausteine (FBs)

SubApp-Instanz	Typ	Zweck

`Input_I1` … `Input_I8`	`MyLib::sys::logiBUS_IXA_BG`	Digitaler Eingang mit VT-Statusanzeige (Grün/Weiß), **ohne** OPC-UA-Publish
`Output_Q1` … `Output_Q12`	`MyLib::sys::Button_IXA_TO_logiBUS_QXA_BG`	Digitaler Ausgang, schaltbar über VT-Taster, **ohne** OPC-UA-Subscribe

Sub-Baustein: `logiBUS_IXA_BG` (Eingänge)

- **Typ**: SubAppType (`MyLib::sys`)
- **Funktionsweise**: Liest einen physischen digitalen Eingang (`logiBUS_IXA`) und zeigt den Zustand direkt über `GreenWhiteBackground1_AX` als VT-Hintergrundfarbe (Grün/Weiß) an. Kein Adapter-Split, kein OPC-UA-Zweig — die einfachste Stufe dieser Eingangsfamilie.

Sub-Baustein: `Button_IXA_TO_logiBUS_QXA_BG` (Ausgänge)

- **Typ**: SubAppType (`MyLib::sys`)
- **Funktionsweise**: Ein VT-Taster (`Button_IXA`) schaltet über ein `AX_SR`-Flipflop den physischen Ausgang (`logiBUS_QXA`) und dessen VT-Statusfarbe (`GreenWhiteBackground1_AX`). Nur eine Schaltquelle (VT), kein OPC-UA-Subscribe und damit auch keine der bei der OPC-UA-Variante nötigen Feedback-Loop-Entkopplung.

Programmablauf und Verbindungen

Die Übung selbst enthält **keine Verbindungen** (`SubAppNetwork` besteht nur aus SubApp-Instanzen mit Parametern):

1. **8 Eingänge**: `Input_I1`…`Input_I8` lesen `Input_I1`…`Input_I8` und spiegeln sie per VT-Statusfarbe.
2. **12 Ausgänge**: `Output_Q1`…`Output_Q12` verbinden je einen physischen Ausgang mit VT-Taster und VT-Statusfarbe.

Registrierung im Trainingssystem: Wie bei allen Übungen in diesem System kein eigenes `Application`-Element nötig — Auswahl per „Change Type“ im 4diac IDE auf den einen `Control`-Slot des `System`.

Lernziele

- Grundmuster für digitale Ein-/Ausgänge rein über den VT, als Ausgangspunkt vor der OPC-UA-Erweiterung.
- Direkter Vergleich mit `InputOutputTesterButton_DIDO_OPC_UA` zeigt genau, was für eine bidirektionale Web-Anbindung zusätzlich nötig ist (Adapter-Splits, `SystemTickSender`, Feedback-Loop-Entkopplung).

Schwierigkeitsgrad: Einsteiger **Vorkenntnisse**: Grundlagen der logiBUS-Digital-I/O-Bausteine (`logiBUS_IXA`, `logiBUS_QXA`).

Zusammenfassung

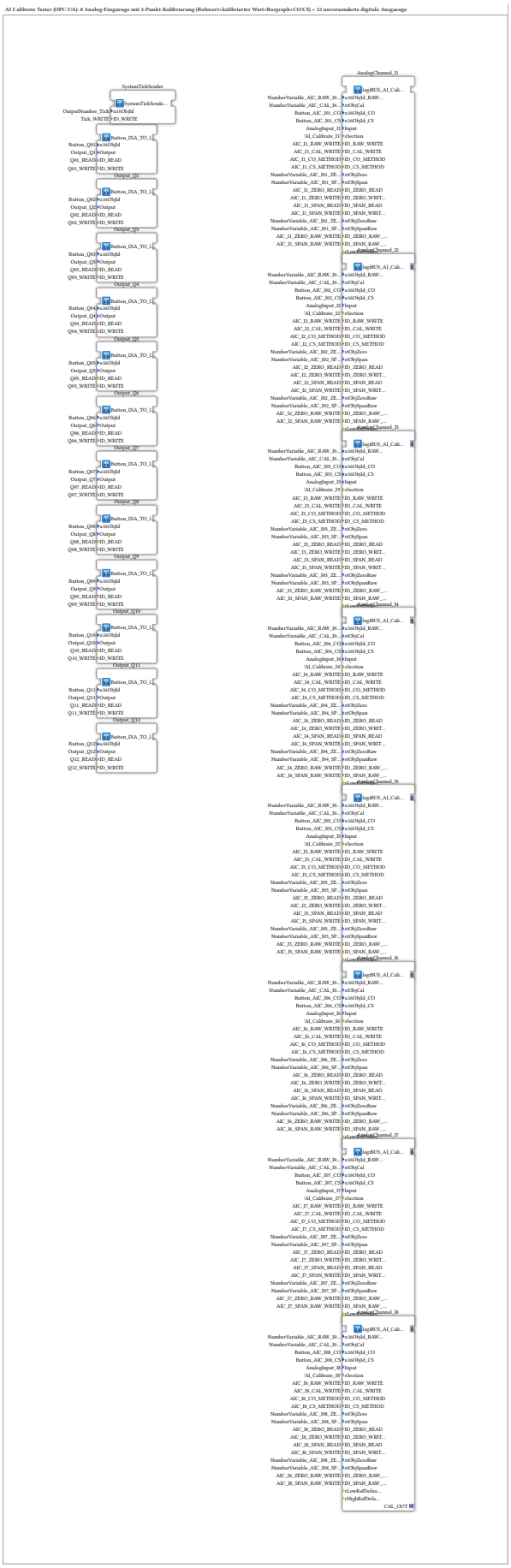
`InputOutputTesterBt` ist die reine VT-Basisversion des DIDO-Testers: 8 Eingänge, 12 Ausgänge, keine OPC-UA-Anbindung. Dient als didaktischer Ausgangspunkt und Vergleichsbasis für die spätere OPC-UA-Erweiterung.

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

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

Button AI Calibrate 2P OPC UA

InputOutputTesterButton_AI_Calibrate_2P_OPC_UA: AI Calibrate Tester (OPC-UA), 2-Punkt


AI Calibrate Tester (OPC-UA): 8 Analog-Eingaenge mit 2-Punkt-Kalibrierung (Rohwert+kalibrierter Wert+Bargraphs+CO/CS) + 12 unveraenderte digitale Ausgaenge
SystemTickSender
Button_IXA_TO_1
Output_Q0
Output_Q1
Output_Q2
Output_Q3
Output_Q4
Output_Q5
Output_Q6
Output_Q7
Output_Q8
Output_Q9
Output_Q10
Output_Q11
Output_Q12
AnalogChannel_I1
AnalogChannel_I2
AnalogChannel_I3
AnalogChannel_I4
AnalogChannel_I5
AnalogChannel_I6
AnalogChannel_I7
AnalogChannel_I8
CAL_OUT

• – ◦ • – ◦ • – ◦ •

Einleitung

`InputOutputTesterButton_AI_Calibrate_2P_OPC_UA` ist das Trainingsbeispiel für **8 analoge Eingänge mit 2-Punkt-Kalibrierung** (Nullpunkt/Spanne), bedienbar sowohl über den ISOBUS-Virtual-Terminal als auch über OPC-UA (Web-Client). Jeder Kanal zeigt Rohwert, kalibrierten Wert und Bargraph an; die Kalibrierung wird mit zwei Tasten (CO = Nullpunkt, CS = Spanne) ausgelöst. Die 12 digitalen Ausgänge sind unverändert aus dem `InputOutputTesterButton_DIDO_OPC_UA`-Beispiel übernommen.

Gegenüber dem Vorgänger `InputOutputTesterButton_AI_Calibrate_OPC_UA` wurde die Kalibrierlogik umgebaut: Statt `AR_CALIBRATE_SQ_REF` mit OFFSET/SCALE arbeitet der Kanalbaustein mit `AR_CALIBRATE_2P_REF`, das die beiden Rohwerte (`X_LOW`/`X_HIGH`) und die beiden Referenzwerte (`LOW_REF`/`HIGH_REF`) unabhängig voneinander speichert. Eine feste Reihenfolge CO vor CS ist daher nicht mehr nötig. Die Fernauslösung über OPC-UA erfolgt nicht mehr über Toggle-Knoten (`AX_SUBSCRIBE`/`AX_RF_TRIG`), sondern über argumentlose OPC-UA-**Methodenaufrufe** (`ID_CO_METHOD`/`ID_CS_METHOD`), die als Ereignis direkt an die Kalibrierung gehen.

Verwendete Funktionsbausteine (FBs)

Instanz	Typ	Parameter	Erklärung
`AnalogChannel_I1` … `AnalogChannel_I8`	`MyLib::sys::logiBUS_AI_Calibrate_2P_IDA_OPC`	`Input` = `AnalogInput_I1`…`I8`; `sSection` = `'AI_Calibrate_I1'`…`'AI_Calibrate_I8'`; VT-Objekte `NumberVariable_AIC_RAW_I0n`, `NumberVariable_AIC_CAL_I0n`, `Button_AIC_I0n_CO`/`_CS`; OPC-UA-IDs `AIC_In_RAW_WRITE`, `AIC_In_CAL_WRITE`, `AIC_In_CO_METHOD`, `AIC_In_CS_METHOD`	Analogeingang mit 2-Punkt-Kalibrierung, VT-Anzeige (Rohwert, kalibriert, Bargraph) und OPC-UA-Anbindung
`Output_Q1` … `Output_Q12`	`MyLib::sys::Button_IXA_TO_logiBUS_QXA_BG_OPC`	`u16ObjId` = `Button_Q01`…`Q12`; `Output` = `Output_Q1`…`Q12`; `ID_READ`/`ID_WRITE` = `Q01_READ`/`Q01_WRITE` …	Digitaler Ausgang, unverändert aus dem DIDO-Beispiel
`SystemTickSender`	`MyLib::sys::SystemTickSender_ISO_OPC`	`u16ObjId` = `OutputNumber_Tick`; `ID_WRITE` = `Tick_WRITE`	Zyklischer Zähler als Lebenszeichen, auf VT-Zahlenfeld und per OPC-UA veröffentlicht

Zusätzliche Parameter der Kanalbausteine

Jeder `AnalogChannel_In` erhält außerdem die Referenzwerte und Rohwert-Diagnose als VT-Objekte und OPC-UA-Knoten:

- `stObjZero` = `NumberVariable_AIC_I0n_ZERO_N`, `stObjSpan` = `NumberVariable_AIC_I0n_SPAN_N`: VT-Eingabefelder für die Referenzwerte (Sollwert am Nullpunkt bzw. an der Spanne).
- `ID_ZERO_READ`/`ID_ZERO_WRITE`, `ID_SPAN_READ`/`ID_SPAN_WRITE`: OPC-UA-Knoten zum Setzen (Subscribe) und Zurückmelden (Publish) der Referenzwerte, z. B. `AIC_I1_ZERO_READ`.
- `stObjZeroRaw` = `NumberVariable_AIC_I0n_ZERORAW_N`, `stObjSpanRaw` = `NumberVariable_AIC_I0n_SPANRAW_N` sowie `ID_ZERO_RAW_WRITE`/`ID_SPAN_RAW_WRITE`: reine Anzeige der bei CO bzw. CS gespeicherten Rohwerte (VT und OPC-UA).

Innenleben des Kanalbausteins

Laut Bibliotheksbaustein `logiBUS_AI_Calibrate_2P_IDA_OPC` (Version 1.x) besteht ein Kanal aus:

- `InputRaw` (`logiBUS_AI_Raw_VT_AR_OPC`): liest den Analogeingang, zeigt den Rohwert an und veröffentlicht ihn.

- `CALIBRATE` (`AR_CALIBRATE_2P_REF`, `QI=TRUE`, `xClipping=FALSE`): eigentliche 2-Punkt-Kalibrierung mit den Ereigniseingängen `C_LOW`/`C_HIGH`.
- `TriggerCO`, `TriggerCS` (`Button_Method_IE`): lösen die Kalibrierung durch die lokale VT-Taste **oder** den OPC-UA-Methodenaufruf aus.
- `Y_OFFSET_LIT`, `Y_SCALE_LIT` (`NumericValue_TO_AR2_OPC`): liefern die Referenzwerte aus VT-Eingabefeld bzw. OPC-UA.
- `INI_X_LOW`, `INI_X_HIGH`, `INI_LOW_REF`, `INI_HIGH_REF` (`INI_AR2`): persistieren die vier Kalibrierwerte unter `sSection` in der INI-Datei, sodass sie einen Neustart überstehen.
- `PublishCal`, `PublishZeroRaw`, `PublishSpanRaw` (`AR_TO_Q_Publish_OPC`): VT-Anzeige und OPC-UA-Publish von kalibriertem Wert und Rohwerten an den Kalibrierpunkten.

Programmablauf und Verbindungen

Das SubApp-Netzwerk enthält **keine** Ereignis- oder Datenverbindungen zwischen den Instanzen: Die 21 Instanzen arbeiten unabhängig voneinander und werden ausschließlich über ihre Parameter (VT-Objekt-IDs, `AnalogInput_*`/`Output_*`-Konstanten, OPC-UA-Adressen) mit der Außenwelt verbunden.

1. **8 Analogkanäle**: `AnalogChannel_I1`…`AnalogChannel_I8` lesen `AnalogInput_I1`…`AnalogInput_I8`. CO speichert den aktuellen Rohwert als Nullpunkt, CS als Spanne; ausgelöst per VT-Taste oder OPC-UA-Methode (`AIC_I*_CO_METHOD`/`AIC_I*_CS_METHOD`). Rohwert und kalibrierter Wert gehen an VT und per OPC-UA (`AIC_I*_RAW_WRITE`/`AIC_I*_CAL_WRITE`) nach außen.
2. **12 digitale Ausgänge**: `Output_Q1`…`Output_Q12`, unverändert aus dem DIDO-Beispiel.
3. **Zeitgeber**: `SystemTickSender` speist das VT-Zahlenfeld `OutputNumber_Tick` und den OPC-UA-Knoten `Tick_WRITE`.

Zusammenfassung

`InputOutputTesterButton_AI_Calibrate_2P_OPC_UA` zeigt die 2-Punkt-Kalibrierung von 8 Analogeingängen mit `AR_CALIBRATE_2P_REF`: unabhängig gespeicherte Rohwerte und Referenzwerte (INI-persistent), Auslösung von CO/CS wahlweise lokal per VT-Taste oder per OPC-UA-Methodenaufruf, Referenzwerte und Rohwert-Diagnose über VT und OPC-UA. Das Beispiel besteht aus nur drei Bausteintypen ohne Verbindungen im Netzwerk, die Logik steckt vollständig im Kanalbaustein.

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

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

Button AI Calibrate 3P OPC UA

InputOutputTesterButton_AI_Calibrate_3P_OPC_UA: AI Calibrate Tester (OPC-UA), 3-Punkt


AI Calibrate 8P Tester (SPC-UA) 8 Analog Eingaenge mit 3-Punkt-Kalibrierung (Rohwert+kalibrierter Wert+Bargraph+C_MIN/C_MID/C_MAX, fuer Sensoren mit Mittelstellung) + 32 nervensaemerte digitale
SystemTickGenerator
SystemTickSchreiber
OutputNumber_Tick
Tick_WRITE
SD_WRITE
Button_DIA_TO_1
Output_Q01
Q01_READ
Q01_WRITE
Output_Q02
Output_Q03
Output_Q04
Output_Q05
Output_Q06
Output_Q07
Output_Q08
Output_Q09
Output_Q10
Output_Q11
Output_Q12
AnalogChannel_01
AnalogChannel_02
AnalogChannel_03
AnalogChannel_04
AnalogChannel_05
AnalogChannel_06
AnalogChannel_07
AnalogChannel_08

• – ◦ • – ◦ • – ◦ •

Einleitung

`InputOutputTesterButton_AI_Calibrate_3P_OPC_UA` ist das Trainingsbeispiel für **8 analoge Eingänge mit 3-Punkt-Kalibrierung** (Minimum/Mitte/Maximum), gedacht für Sensoren mit Mittelstellung, z. B. einen Lenkwinkelsensor (links/geradeaus/rechts). Bedienung über den ISOBUS-Virtual-Terminal und über OPC-UA (Web-Client). Jeder Kanal zeigt Rohwert, kalibrierten Wert und Bargraph; kalibriert wird mit drei Tasten (C_MIN, C_MID, C_MAX). Die 12 digitalen Ausgänge sind unverändert aus dem `InputOutputTesterButton_DIDO_OPC_UA`-Beispiel übernommen.

Das Beispiel ist das 3-Punkt-Pendant zu `InputOutputTesterButton_AI_Calibrate_2P_OPC_UA` (Vorgänger: `InputOutputTesterButton_AI_Calibrate_OPC_UA`): Der Kanalbaustein nutzt `AR_CALIBRATE_3P_REF` mit drei unabhängig gespeicherten Rohwerten (`X_MIN/X_MID/X_MAX`) und drei Referenzwerten (`MIN_REF/MID_REF/MAX_REF`). Alle drei Tasten sind auch als argumentlose OPC-UA-**Methodenaufrufe** (`ID_MIN_METHOD/ID_MID_TRIGGER_METHOD/ID_MAX_METHOD`) auslösbar.

Verwendete Funktionsbausteine (FBs)

Instanz	Typ	Parameter	Erklärung
`AnalogChannel_I1` … `AnalogChannel_I8`	`MyLib::sys::logiBUS_AI_Calibrate_3P_IDA_OPC`	`Input` = `AnalogInput_I1`…`I8`; `sSection` = `'AI_Calibrate_3P_I1'`…`'AI_Calibrate_3P_I8'`; VT-Objekte `NumberVariable_AIC_Raw_I0n`, `NumberVariable_AIC_CAL_I0n_N`, Tasten `u16ObjId_C_MIN` = `Button_AIC_I0n_C0`, `u16ObjId_C_MID` = `Button_AIC_I0n_MID`, `u16ObjId_C_MAX` = `Button_AIC_I0n_CS`; OPC-UA-IDs `AIC_In_RAW_WRITE`, `AIC_In_CAL_WRITE`, `AIC_In_MIN_METHOD`, `AIC_In_MID_METHOD`, `AIC_In_MAX_METHOD`	Analogeingang mit 3-Punkt-Kalibrierung, VT-Anzeige (Rohwert, kalibriert, Bargraph) und OPC-UA-Anbindung
`Output_Q1` … `Output_Q12`	`MyLib::sys::Button_IXA_TO_logiBUS_QXA_BG_OPC`	`u16ObjId` = `Button_Q01`…`Q12`; `Output` = `Output_Q1`…`Q12`; `ID_READ/ID_WRITE` = `Q01_READ/Q01_WRITE` …	Digitaler Ausgang, unverändert aus dem DIDO-Beispiel
`SystemTickSender`	`MyLib::sys::SystemTickSender_ISO_OPC`	`u16ObjId` = `OutputNumber_Tick`; `ID_WRITE` = `Tick_WRITE`	Zyklischer Zähler als Lebenszeichen, auf VT-Zahlenfeld und per OPC-UA veröffentlicht

Zusätzliche Parameter der Kanalbausteine

- `stObjMinRef` = `NumberVariable_AIC_I0n_ZERO_N`, `stObjMidRef` = `NumberVariable_AIC_I0n_MID_N`, `stObjMaxRef` = `NumberVariable_AIC_I0n_SPAN_N`: VT-Eingabefelder für die drei Referenzwerte.
- `ID_MINREF_READ/_WRITE`, `ID_MIDREF_READ/_WRITE`, `ID_MAXREF_READ/_WRITE` (z. B. `AIC_I1_MINREF_READ`): OPC-UA-Knoten zum Setzen (Subscribe) und Zurückmelden (Publish) der Referenzwerte.
- `stObjMinRaw/stObjMidRaw/stObjMaxRaw` = `NumberVariable_AIC_I0n_MINRAW_N/_MIDRAW_N/_MAXRAW_N` (plus je ein `ID_*_RAW_WRITE`-Knoten): reine Anzeige der an den drei Kalibrierpunkten gespeicherten Rohwerte.

Innenleben des Kanalbausteins

Laut Bibliotheksbaustein `logiBUS_AI_Calibrate_3P_IDA_OPC`:

- `InputRaw` (`logiBUS_AI_Raw_VT_AR_OPC`): liest den Analogeingang, zeigt den Rohwert an und veröffentlicht ihn.
- `CALIBRATE` (`AR_CALIBRATE_3P_REF`, `QI=TRUE`, `xClipping=FALSE`): 3-Punkt-Kalibrierung mit den Ereigniseingängen `C_MIN`/`C_MID`/`C_MAX`.
- `TriggerMIN`, `TriggerMID`, `TriggerMAX` (`Button_Method_IE`): lösen per lokaler VT-Taste **oder** OPC-UA-Methodenaufruf aus; alle drei sind baugleich.
- `MIN_REF_LIT`, `MID_REF_LIT`, `MAX_REF_LIT` (`NumericValue_TO_AR2_OPC`): Referenzwerte aus VT-Eingabefeld bzw. OPC-UA.
- `INI_X_MIN`/`INI_X_MID`/`INI_X_MAX` (Vorgabe `0.0`) und `INI_MIN_REF`/`INI_MID_REF`/`INI_MAX_REF` (Vorgaben `-100.0`/`0.0`/`100.0`), alle `INI_AR2`: persistieren die sechs Eckwerte unter `sSection`.
- `PublishCal`, `PublishMinRaw`, `PublishMidRaw`, `PublishMaxRaw` (`AR_TO_Q_Publish_OPC`): VT-Anzeige und OPC-UA-Publish.

Programmablauf und Verbindungen

Das SubApp-Netzwerk enthält **keine** Ereignis- oder Datenverbindungen zwischen den Instanzen: Die 21 Instanzen arbeiten unabhängig voneinander und werden ausschließlich über ihre Parameter (VT-Objekt-IDs, `AnalogInput_*`/`Output_*`-Konstanten, OPC-UA-Adressen) mit der Außenwelt verbunden.

1. **8 Analogkanäle**: `AnalogChannel_I1`…`AnalogChannel_I8` lesen `AnalogInput_I1`…`AnalogInput_I8`. Der Anwender fährt Minimum, Mitte und Maximum an und drückt jeweils C_MIN, C_MID bzw. C_MAX (VT-Taste oder OPC-UA-Methode `AIC_I*_MIN_METHOD`/`_MID_METHOD`/`_MAX_METHOD`). Rohwert und kalibrierter Wert gehen an VT und per OPC-UA (`AIC_I*_RAW_WRITE`/`AIC_I*_CAL_WRITE`) nach außen.
2. **12 digitale Ausgänge**: `Output_Q1`…`Output_Q12`, unverändert aus dem DIDO-Beispiel.
3. **Zeitgeber**: `SystemTickSender` speist das VT-Zahlenfeld `OutputNumber_Tick` und den OPC-UA-Knoten `Tick_WRITE`.

Zusammenfassung

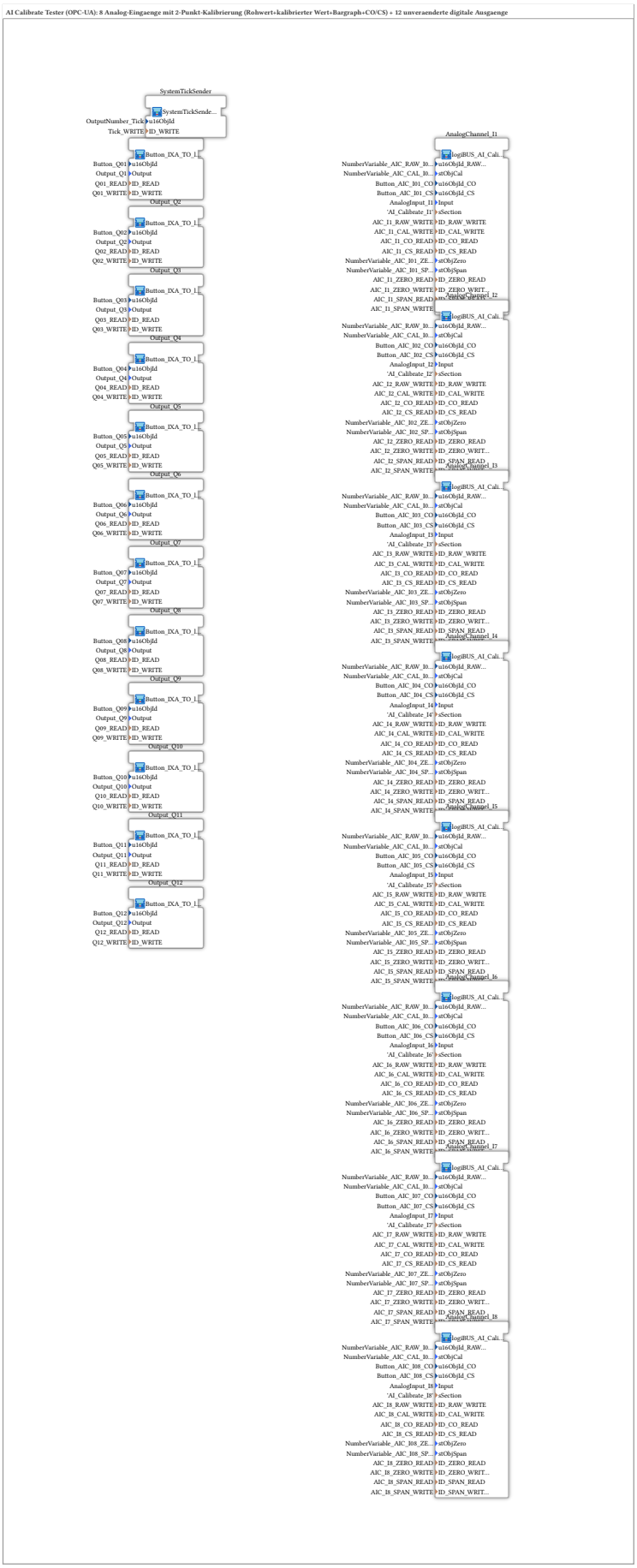
`InputOutputTesterButton_AI_Calibrate_3P_OPC_UA` zeigt die 3-Punkt-Kalibrierung von 8 Analogeingängen mit `AR_CALIBRATE_3P_REF` für Sensoren mit Mittelstellung: drei unabhängig gespeicherte Rohwerte und Referenzwerte (INI-persistent), Auslösung wahlweise per VT-Taste oder OPC-UA-Methodenaufruf, Referenzwerte und Rohwert-Diagnose über VT und OPC-UA. Das Netzwerk enthält keine Verbindungen, die Logik steckt vollständig im Kanalbaustein.

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

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

Button AI Calibrate OPC UA

InputOutputTesterButton_AI_Calibrate_OPC_UA: AI Calibrate Tester (OPC-UA)


AI Calibrate Tester (OPC-UA): 8 Analog-Eingaenge mit 2-Punkt-Kalibrierung (Rohwert+kalibrierter Wert+Bargraph+CO/CS) + 12 unveraenderte digitale Ausgaenge
SystemTickSender
SystemTickSende...
OutputNumber_Tick
Tick_WRITE
u16ObjId
ID_WRITE
Button_IXA_TO_1...
Button_Q01
Output_Q1
Q01_READ
Q01_WRITE
Output_Q2
Button_Q02
Output_Q2
Q02_READ
Q02_WRITE
Output_Q3
Button_Q03
Output_Q3
Q03_READ
Q03_WRITE
Output_Q4
Button_Q04
Output_Q4
Q04_READ
Q04_WRITE
Output_Q5
Button_Q05
Output_Q5
Q05_READ
Q05_WRITE
Output_Q6
Button_Q06
Output_Q6
Q06_READ
Q06_WRITE
Output_Q7
Button_Q07
Output_Q7
Q07_READ
Q07_WRITE
Output_Q8
Button_Q08
Output_Q8
Q08_READ
Q08_WRITE
Output_Q9
Button_Q09
Output_Q9
Q09_READ
Q09_WRITE
Output_Q10
Button_Q10
Output_Q10
Q10_READ
Q10_WRITE
Output_Q11
Button_Q11
Output_Q11
Q11_READ
Q11_WRITE
Output_Q12
Button_Q12
Output_Q12
Q12_READ
Q12_WRITE
AnalogChannel_I1
AnalogChannel_I2
AnalogChannel_I3
AnalogChannel_I4
AnalogChannel_I5
AnalogChannel_I6
AnalogChannel_I7
AnalogChannel_I8
LogiBUS_AI_Cali...
NumberVariable_AIC_RAW_I0...
NumberVariable_AIC_CAL_I0...
Button_AIC_I01_CO
Button_AIC_I01_CS
AnalogInput_I1
'AI_Calibrate_I1'
AIC_I1_RAW_WRITE
AIC_I1_CAL_WRITE
AIC_I1_CO_READ
AIC_I1_CS_READ
NumberVariable_AIC_I01_ZE...
NumberVariable_AIC_I01_SP...
AIC_I1_ZERO_READ
AIC_I1_ZERO_WRITE
AIC_I1_SPAN_READ
AIC_I1_SPAN_WRITE
Button_AIC_I02_CO
Button_AIC_I02_CS
AnalogInput_I2
'AI_Calibrate_I2'
AIC_I2_RAW_WRITE
AIC_I2_CAL_WRITE
AIC_I2_CO_READ
AIC_I2_CS_READ
NumberVariable_AIC_I02_ZE...
NumberVariable_AIC_I02_SP...
AIC_I2_ZERO_READ
AIC_I2_ZERO_WRITE
AIC_I2_SPAN_READ
AIC_I2_SPAN_WRITE
Button_AIC_I03_CO
Button_AIC_I03_CS
AnalogInput_I3
'AI_Calibrate_I3'
AIC_I3_RAW_WRITE
AIC_I3_CAL_WRITE
AIC_I3_CO_READ
AIC_I3_CS_READ
NumberVariable_AIC_I03_ZE...
NumberVariable_AIC_I03_SP...
AIC_I3_ZERO_READ
AIC_I3_ZERO_WRITE
AIC_I3_SPAN_READ
AIC_I3_SPAN_WRITE
Button_AIC_I04_CO
Button_AIC_I04_CS
AnalogInput_I4
'AI_Calibrate_I4'
AIC_I4_RAW_WRITE
AIC_I4_CAL_WRITE
AIC_I4_CO_READ
AIC_I4_CS_READ
NumberVariable_AIC_I04_ZE...
NumberVariable_AIC_I04_SP...
AIC_I4_ZERO_READ
AIC_I4_ZERO_WRITE
AIC_I4_SPAN_READ
AIC_I4_SPAN_WRITE
Button_AIC_I05_CO
Button_AIC_I05_CS
AnalogInput_I5
'AI_Calibrate_I5'
AIC_I5_RAW_WRITE
AIC_I5_CAL_WRITE
AIC_I5_CO_READ
AIC_I5_CS_READ
NumberVariable_AIC_I05_ZE...
NumberVariable_AIC_I05_SP...
AIC_I5_ZERO_READ
AIC_I5_ZERO_WRITE
AIC_I5_SPAN_READ
AIC_I5_SPAN_WRITE
Button_AIC_I06_CO
Button_AIC_I06_CS
AnalogInput_I6
'AI_Calibrate_I6'
AIC_I6_RAW_WRITE
AIC_I6_CAL_WRITE
AIC_I6_CO_READ
AIC_I6_CS_READ
NumberVariable_AIC_I06_ZE...
NumberVariable_AIC_I06_SP...
AIC_I6_ZERO_READ
AIC_I6_ZERO_WRITE
AIC_I6_SPAN_READ
AIC_I6_SPAN_WRITE
Button_AIC_I07_CO
Button_AIC_I07_CS
AnalogInput_I7
'AI_Calibrate_I7'
AIC_I7_RAW_WRITE
AIC_I7_CAL_WRITE
AIC_I7_CO_READ
AIC_I7_CS_READ
NumberVariable_AIC_I07_ZE...
NumberVariable_AIC_I07_SP...
AIC_I7_ZERO_READ
AIC_I7_ZERO_WRITE
AIC_I7_SPAN_READ
AIC_I7_SPAN_WRITE
Button_AIC_I08_CO
Button_AIC_I08_CS
AnalogInput_I8
'AI_Calibrate_I8'
AIC_I8_RAW_WRITE
AIC_I8_CAL_WRITE
AIC_I8_CO_READ
AIC_I8_CS_READ
NumberVariable_AIC_I08_ZE...
NumberVariable_AIC_I08_SP...
AIC_I8_ZERO_READ
AIC_I8_ZERO_WRITE
AIC_I8_SPAN_READ
AIC_I8_SPAN_WRITE

• – ◦ • – ◦ • – ◦ •

Einleitung

`InputOutputTesterButton_AI_Calibrate_OPC_UA` ist das Trainingsbeispiel für **8 analoge Eingänge mit 2-Punkt-Kalibrierung**, steuerbar sowohl über den ISOBUS-Virtual-Terminal als auch über OPC-UA (Web-Client). Die 12 digitalen Ausgänge sind unverändert aus dem `InputOutputTesterButton_DIDO_OPC_UA`-Beispiel übernommen; neu ist ausschließlich die Kalibrierlogik für die 8 Analogeingänge.

Anders als bei DIDO/PWM ist die eigentliche Kalibrierlogik pro Kanal deutlich umfangreicher: Jeder Analogeingang durchläuft eine vollständige 2-Punkt-Kalibrierung (Referenzwerte ZERO/SPAN, ausgelöst über zwei VT-/OPC-UA-Tasten CO/CS), deren Ergebnis (OFFSET/SCALE) INI-persistiert wird, sodass die Kalibrierung einen Neustart der Steuerung übersteht.

Verwendete Funktionsbausteine (FBs)

SubApp-Instanz	Typ	Zweck
`AnalogChannel_I1` … `AnalogChannel_I8`	`MyLib::sys::logiBUS_AI_Calibrate_IDA_OPC`	Analogeingang mit 2-Punkt-Kalibrierung, VT-Zahlenfeldern (Rohwert + kalibriert) + Bargraph + OPC-UA
`Output_Q1` … `Output_Q12`	`MyLib::sys::Button_IXA_TO_logiBUS_QXA_BG_OPC`	Digitaler Ausgang, unverändert aus dem DIDO-Beispiel übernommen
`SystemTickSender`	`MyLib::sys::SystemTickSender`	Zyklischer Zähler für die VT-Statusanzeige (`OutputNumber_Tick`)

Sub-Baustein: `logiBUS_AI_Calibrate_IDA_OPC` (Analogeingänge)

Noch nicht in der Bibliotheksreferenz dokumentiert (`MyLib_AX-1.0.0::sys`) — separate Doku-Lücke, siehe Hinweis am Ende dieser Seite.

- **Typ**: SubAppType (`MyLib::sys`), Kommentar im Quellbaustein: *„logiBUS_AI_IDA (Analog-Eingang) mit 2-Punkt-Kalibrierung (AR_CALIBRATE_SQ_REF, OFFSET/SCALE + Referenzwerte ZERO/SPAN alle INI-persistiert) auf VT-Zahlenfelder (Rohwert + kalibrierter Wert, je eigene CNumberVariable) + Bargraph (kalibriert) + 2 VT-Tasten (CO/CS) + OPC-UA (Rohwert DWORD + kalibrierter Wert REAL, generisch, ein Kanal)“*
- **Funktionsweise**: Der physische Analogeingang (`logiBUS_AI_IDA`) liefert über eine Kette aus `AD_SPLIT_2`/`AD_TO_AUDI`/`AUDI_SPLIT_2` sowohl den unkalibrierten Rohwert (VT-Anzeige + OPC-UA-Publish über `AD_PUBLISH_1`) als auch den Eingang für die eigentliche Kalibrierung (`AUDI_TO_AR` → `AR_CALIBRATE_SQ_REF`-Adapter). Zwei VT-/OPC-UA-Tasten (`Button_CO`, `Button_CS`) lösen — jeweils über `E_MERGE`-Event-Zusammenführung von lokalem Tastendruck und OPC-UA-Fernauslösung (`AX_SUBSCRIBE_1` → `AX_RF_TRIG`, Flankenerkennung für den Toggle-Mechanismus des Web-Clients) — die beiden Kalibrierschritte CO (Nullpunkt) und CS (Spanne) am `CALIBRATE`-Adapter aus. Das Ergebnis (`OFFSET`, `SCALE`) wird über je einen `INI_AR2`-Baustein persistiert (Default `OFFSET=0.0`, `SCALE=1.0`), ebenso die beiden Referenzwerte `ZERO`/`SPAN` (Default `0.0`/`100.0`). Der kalibrierte Wert (`CALIBRATE.Y`) geht über `AR_SPLIT_2` sowohl an die VT-/OPC-UA-Anzeige (`AR_PUBLISH_1`) als auch an das VT-Zahlenfeld mit Bargraph (`Q_NumericValue_PHYSA`).
- **Referenzwerte von außen setzbar**: Die ZERO-/SPAN-Referenzwerte selbst sind über zwei `MyLib::sys::NumericValue_TO_AR2_OPC`-Sub-Bausteine (`Y_OFFSET_LIT`, `Y_SCALE_LIT` — Namen historisch aus einer früheren Iteration, liefern tatsächlich die ZERO/SPAN-Werte an `CALIBRATE.Y_Offset`/`CALIBRATE.Y_Scale`) sowohl per VT-Eingabefeld als auch per OPC-UA schreibbar — anders als bei DIDO/PWM sind hier also nicht nur Ausgänge, sondern auch Kalibrierparameter bidirektional von VT und Web aus änderbar.
- **Kalibrierverfahren**: Verwendet `AR_CALIBRATE_SQ_REF` statt des einfacheren `AR_CALIBRATE` (siehe `AR_CALIBRATE_SQ` in der Bibliotheksreferenz) — laut Baustein-Versionshinweis nach Vorbild einer früheren Übung, aber mit korrigierter Offset-Formel und ECC-erzwungener CO-vor-CS-Reihenfolge (Spanne kann erst nach dem Nullpunkt kalibriert werden).

Sub-Baustein: Button_IXA_TO_logiBUS_QXA_BG_OPC (Ausgänge)

Unverändert aus dem DIDO-Beispiel — siehe dortige Beschreibung.

VT-Anbindung (ISOBUS Virtual Terminal)

Das zugehörige VT-Pool-Projekt `Workspace_AI_Calibrate` (ISO-Designer, `DefaultPool.iop`) besteht aus 5 DataMasken:

Maske	ObjectID	Inhalt
`DataMask_M1`	1000	Übersichts-/Startmaske (480×480), Zugriffspunkt für die 4 Kanal-Masken
`DataMask_AIC1` … `DataMask_AIC4`	1001–1004	je **2 Analogkanäle**: Rohwert-Anzeige, kalibrierter Wert, Bargraph, 2 CO/CS-Tasten und 2 Eingabefelder für die ZERO-/SPAN-Referenzwerte pro Kanal

Die numerischen VT-Objekte (`NumberVariable_AIC_RAW_I0n`, `NumberVariable_AIC_CAL_I0n_N`, `NumberVariable_AIC_I0n_ZERO_N/_SPAN_N`, `Button_AIC_I0n_CO/_CS`) werden — wie bei allen VT-Übungen in diesem System — über `GcfScript.py` aus dem kompilierten `DefaultPool.iop.h` in eine 4diac-`.gcf`-Konstantendatei (`Uebungen::const::UT::AIC::DefaultPool_AIC`) übersetzt und dort per `Import` in die FB-Netzwerke eingebunden (siehe `test_AX/scripts/RunSkript_Workspace_AI_Calibrate_AX.{bat,sh}`).

Web-Client (OPC-UA, kein VT-Pixel-Mirror)

Der Web-Client `apixon-ai-calibrate-client` (Vue 3, `ApixonAICalibrate.vue`) ist **kein** visueller Nachbau der DataMasken (anders als ein `vt-ui-mirror`-Projekt), sondern eine eigenständige, funktionale Web-Oberfläche: Er verbindet sich direkt per WebSocket-OPC-UA (`OPCUAClient`, Standardport `4841`) mit dem FORTE-Laufzeitsystem und abonniert/schreibt dieselben Knoten wie die FB-Netzwerke (Rohwert, kalibrierter Wert, CO-/CS-Toggle, ZERO/SPAN-Referenzwerte je Kanal) — unabhängig vom Layout der VT-Masken.

Programmablauf und Verbindungen

1. **8 Analogkanäle**: `AnalogChannel_I1`…`AnalogChannel_I8` lesen `AnalogInput_I1`…`AnalogInput_I8`, kalibrieren sie 2-Punkt-basiert und veröffentlichen Rohwert + kalibrierten Wert per OPC-UA (`AIC_I1_RAW_WRITE`…`AIC_I8_CAL_WRITE`); die CO-/CS-Tasten sind sowohl lokal (VT) als auch remote (OPC-UA, `AIC_I*_CO_READ/_CS_READ`) auslösbar.
2. **12 digitale Ausgänge**: `Output_Q1`…`Output_Q12`, unverändert aus dem DIDO-Beispiel.
3. **Zeitgeber**: `SystemTickSender` speist das VT-Zahlenfeld `OutputNumber_Tick` sowie den OPC-UA-Knoten `Tick_WRITE`.

Registrierung im Trainingssystem: Wie bei allen Übungen in diesem System kein eigenes `Application`-Element nötig — Auswahl per „Change Type“ im 4diac IDE auf den einen `Control`-Slot des `System`.

Lernziele

- 2-Punkt-Kalibrierung (`AR_CALIBRATE_SQ_REF`) eines Analogeingangs mit paralleler VT- **und** OPC-UA-Bedienung, inklusive INI-Persistenz der Kalibrierparameter über einen Neustart hinweg.
- Bidirektionale Fernauslösung eines lokal ausgelösten Vorgangs (CO/CS-Tasten) über OPC-UA mittels Flankenerkennung (`AX_RF_TRIG`) statt direkter Zustandsübernahme — Toggle-Semantik des Web-Clients.
- Kombination aus reinem Publish (Rohwert), bidirektionalem Trigger (CO/CS) und bidirektional beschreibbaren Parametern (ZERO/SPAN-Referenzwerte) in einem einzigen Baustein.

Schwierigkeitsgrad: Mittel bis Fortgeschritten **Vorkenntnisse**: `InputOutputTesterButton_DIDO_OPC_UA` (Grundmuster VT+OPC-UA), `AR_CALIBRATE_SQ`-Adapter, INI-Persistenz (`eclipse4diac::storage::INI_AR2`).

Zusammenfassung

`InputOutputTesterButton_AI_Calibrate_OPC_UA` demonstriert 2-Punkt-Analogkalibrierung mit vollem VT- und OPC-UA-Zugriff auf Rohwert, kalibrierten Wert, Kalibrierauslösung und die Referenzwerte selbst — die bisher komplexeste Kombination aus lokaler und ferngesteuerter Bedienung in diesem Trainingssystem.

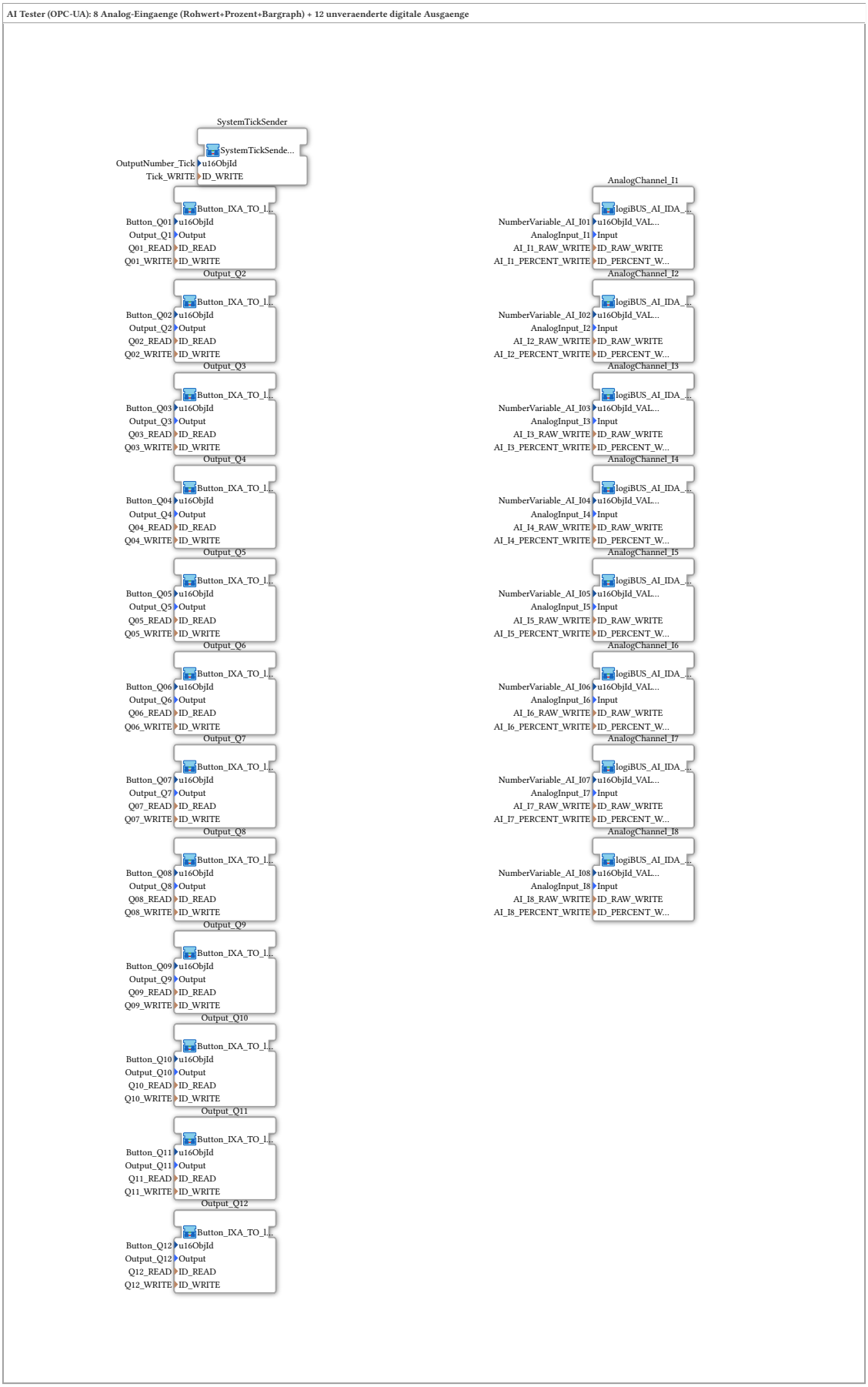
Hinweis: Der Sub-Baustein `MyLib::sys::logiBUS_AI_Calibrate_IDA_OPC` sowie `MyLib::sys::NumericValue_TO_AR2_OPC` liegen unter `MyLib_AX-1.0.0` und haben noch keine eigene Seite in der Bibliotheksreferenz (`Bibliotheken/typelibrary/MyLib_AX/sys/`) — das ist Teil einer separaten, größeren Doku-Lücke für die `MyLib_AX`/`MyLib_B`-Bibliotheksreferenz und nicht Gegenstand dieser Seite.

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

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

Button AI OPC UA

InputOutputTesterButton_AI_OPC_UA: AI Tester (OPC-UA, ohne Kalibrierung)


AI Tester (OPC-UA): 8 Analog-Eingaenge (Rohwert+Prozent+Bargraph) + 12 unveraenderte digitale Ausgaenge
SystemTickSender
SystemTickSende...
OutputNumber_Tick
u16ObjId
Tick_WRITE
ID_WRITE
Button_IXA_TO_I...
Button_Q01
u16ObjId
Output_Q1
Output
Q01_READ
ID_READ
Q01_WRITE
ID_WRITE
Output_Q2
Button_Q02
Output_Q2
Q02_READ
Q02_WRITE
Output_Q3
Button_Q03
Output_Q3
Q03_READ
Q03_WRITE
Output_Q4
Button_Q04
Output_Q4
Q04_READ
Q04_WRITE
Output_Q5
Button_Q05
Output_Q5
Q05_READ
Q05_WRITE
Output_Q6
Button_Q06
Output_Q6
Q06_READ
Q06_WRITE
Output_Q7
Button_Q07
Output_Q7
Q07_READ
Q07_WRITE
Output_Q8
Button_Q08
Output_Q8
Q08_READ
Q08_WRITE
Output_Q9
Button_Q09
Output_Q9
Q09_READ
Q09_WRITE
Output_Q10
Button_Q10
Output_Q10
Q10_READ
Q10_WRITE
Output_Q11
Button_Q11
Output_Q11
Q11_READ
Q11_WRITE
Output_Q12
Button_Q12
Output_Q12
Q12_READ
Q12_WRITE
AnalogChannel_I1
logiBUS_AI_IDA_...
NumberVariable_AI_I01
u16ObjId_VAL...
AnalogInput_I1
Input
AI_I1_RAW_WRITE
ID_RAW_WRITE
AI_I1_PERCENT_WRITE
ID_PERCENT_W...
AnalogChannel_I2
NumberVariable_AI_I02
AnalogInput_I2
AI_I2_RAW_WRITE
AI_I2_PERCENT_WRITE
AnalogChannel_I3
NumberVariable_AI_I03
AnalogInput_I3
AI_I3_RAW_WRITE
AI_I3_PERCENT_WRITE
AnalogChannel_I4
NumberVariable_AI_I04
AnalogInput_I4
AI_I4_RAW_WRITE
AI_I4_PERCENT_WRITE
AnalogChannel_I5
NumberVariable_AI_I05
AnalogInput_I5
AI_I5_RAW_WRITE
AI_I5_PERCENT_WRITE
AnalogChannel_I6
NumberVariable_AI_I06
AnalogInput_I6
AI_I6_RAW_WRITE
AI_I6_PERCENT_WRITE
AnalogChannel_I7
NumberVariable_AI_I07
AnalogInput_I7
AI_I7_RAW_WRITE
AI_I7_PERCENT_WRITE
AnalogChannel_I8
NumberVariable_AI_I08
AnalogInput_I8
AI_I8_RAW_WRITE
AI_I8_PERCENT_WRITE

• – ◦ • – ◦ • – ◦ •

Einleitung

`InputOutputTesterButton_AI_OPC_UA` ist das Trainingsbeispiel für **8 analoge Eingänge (Rohwert + Prozent + Bargraph)**, steuerbar sowohl über den ISOBUS-Virtual-Terminal als auch über OPC-UA. Die 12 digitalen Ausgänge sind unverändert aus `InputOutputTesterButton_DIDO_OPC_UA` übernommen.

Diese Übung ist die **einfachere Vorstufe** zu `InputOutputTesterButton_AI_Calibrate_OPC_UA`: Sie zeigt Rohwert und (linear umgerechneten) Prozentwert eines Analogeingangs, aber **ohne** die spätere 2-Punkt-Kalibrierlogik (kein `CALIBRATE`-Adapter, keine CO-/CS-Tasten, keine INI-persistierten Offset/Scale-Werte, keine remote setzbaren ZERO-/SPAN-Referenzen). Wer verstehen will, was die Kalibrier-Erweiterung an FB-Netzwerk hinzufügt, vergleicht diese beiden Übungen am besten direkt.

Verwendete Funktionsbausteine (FBs)

SubApp-Instanz	Typ	Zweck
`AnalogChannel_I1` … `AnalogChannel_I8`	`MyLib::sys::logiBUS_AI_IDA_OPC`	Analogeingang mit Rohwert- und Prozent-Anzeige (VT-Zahlenfelder + Bargraph) + OPC-UA, **ohne** Kalibrierung
`Output_Q1` … `Output_Q12`	`MyLib::sys::Button_IXA_TO_logiBUS_QXA_BG_OPC`	Digitaler Ausgang, unverändert aus dem DIDO-Beispiel übernommen

Sub-Baustein: `logiBUS_AI_IDA_OPC` (Analogeingänge)

- **Typ**: SubAppType (`MyLib::sys`)
- **Funktionsweise**: Der physische Analogeingang (`logiBUS_AI_IDA`) liefert Rohwert und linear umgerechneten Prozentwert direkt an je ein VT-Zahlenfeld mit Bargraph sowie per OPC-UA-Publish (`ID_RAW_WRITE`, `ID_PERCENT_WRITE`) an den Web-Client. Anders als bei der Calibrate-Variante gibt es keine adaptive Kalibrierkette (`AR_CALIBRATE_SQ_REF`), keine Referenzwerte und keine remote auslösbaren Kalibrierschritte — der Baustein ist reiner Publish-Pfad ohne Rückschreibmöglichkeit.

Sub-Baustein: Button_IXA_TO_logiBUS_QXA_BG_OPC (Ausgänge)

Unverändert aus dem DIDO-Beispiel — siehe dortige Beschreibung.

OPC-UA-Adressraum

Node-Pfad	Node-ID	Bedeutung
`/Objects/Analog/In/RAW`	`s=AI_In_RAW`	Rohwert Eingang n (n=1–8), nur Publish
`/Objects/Analog/In/PERCENT`	`s=AI_In_PERCENT`	Prozentwert Eingang n (n=1–8), nur Publish
`/Objects/DigitalOutput/Qnn`	`s=Qnn`	Ausgang nn (nn=01–12), Read (Subscribe) + Write (Publish/Echo), wie im DIDO-Beispiel

Flacher, nach Kanal getrennter Adressraum, kein verschachtelter Pfad wie beim PWM-Beispiel — analog zum DIDO-Adressraum.

Programmablauf und Verbindungen

Die Übung selbst enthält **keine Verbindungen** (`SubAppNetwork` besteht nur aus SubApp-Instanzen mit Parametern):

1. **8 Analogkanäle**: `AnalogChannel_I1`…`AnalogChannel_I8` lesen `AnalogInput_I1`…`AnalogInput_I8` und veröffentlichen Rohwert + Prozentwert per OPC-UA.
2. **12 digitale Ausgänge**: `Output_Q1`…`Output_Q12`, unverändert aus dem DIDO-Beispiel.

Registrierung im Trainingssystem: Wie bei allen Übungen in diesem System kein eigenes `Application`-Element nötig — Auswahl per „Change Type“ im 4diac IDE auf den einen `Control`-Slot des `System`.

Lernziele

- Grundmuster für einen reinen Publish-Analogeingang (Rohwert + Prozent) mit VT- **und** OPC-UA-Anbindung, ohne die Komplexität einer Kalibrierlogik.
- Direkter Vergleich mit `InputOutputTesterButton_AI_Calibrate_OPC_UA` zeigt konkret, welche zusätzlichen Bausteine (CALIBRATE-Adapter, INI-Persistenz, Referenzwert-Tasten) eine 2-Punkt-Kalibrierung erfordert.

Schwierigkeitsgrad: Einsteiger bis Mittel **Vorkenntnisse**: `InputOutputTesterButton_DIDO_OPC_UA` (Grundmuster VT+OPC-UA).

Zusammenfassung

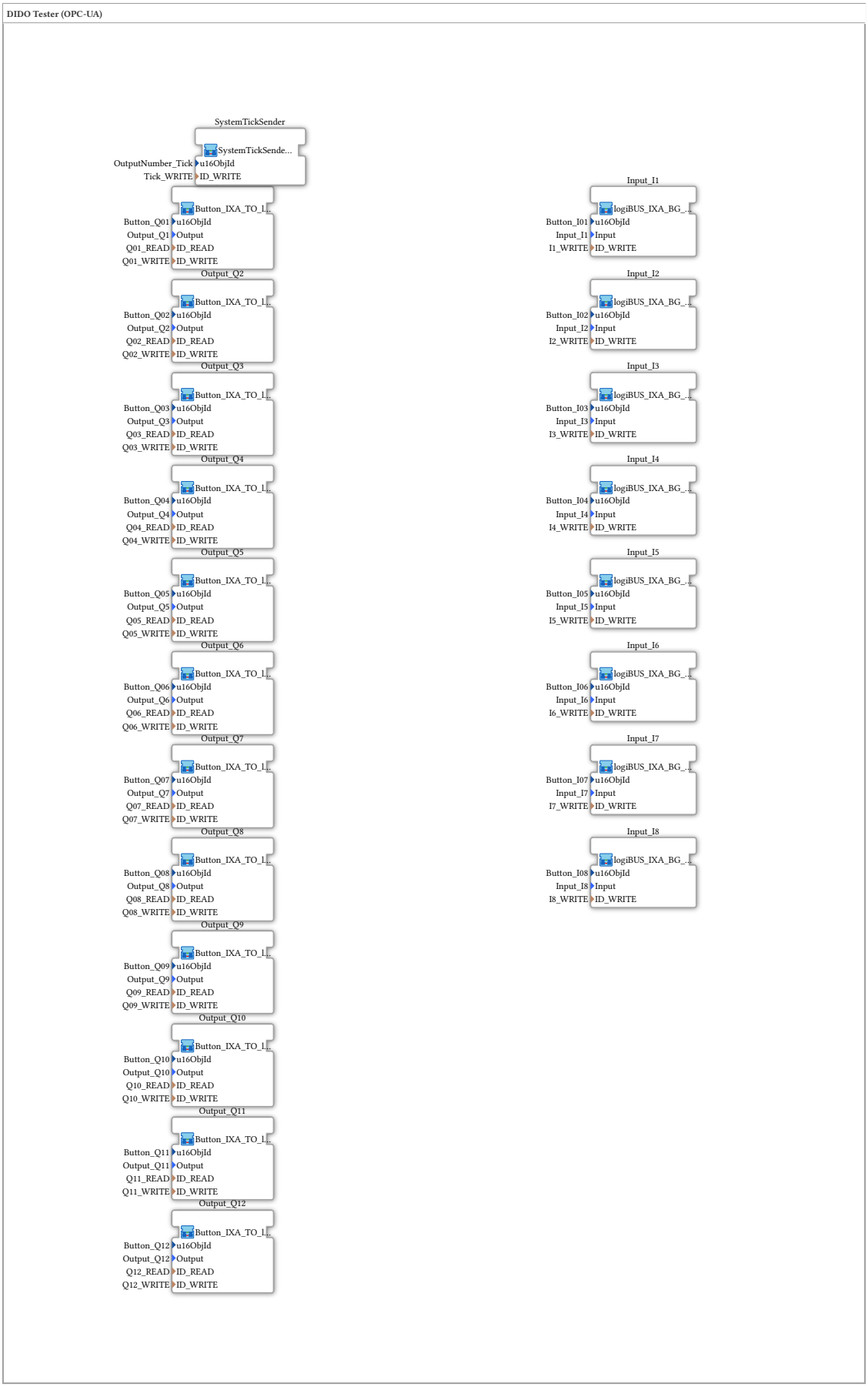
`InputOutputTesterButton_AI_OPC_UA` demonstriert die einfachste Form eines analogen OPC-UA-Eingangs: Rohwert und Prozentwert, rein per Publish, ohne Kalibrierlogik. Bildet die didaktische Brücke zwischen dem rein digitalen DIDO-Beispiel und der deutlich komplexeren 2-Punkt-Kalibrierung.

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

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

Button DIDO OPC UA

InputOutputTesterButton_DIDO_OPC_UA: DIDO Tester (OPC-UA)


DIDO Tester (OPC-UA)
SystemTickSender
SystemTickSende...
OutputNumber_Tick
Tick_WRITE
u16ObjId
ID_WRITE
Button_IXA_TO_I...
Button_Q01
Output_Q1
Q01_READ
Q01_WRITE
Output
ID_READ
Output_Q2
Button_Q02
Output_Q2
Q02_READ
Q02_WRITE
Output_Q3
Button_Q03
Output_Q3
Q03_READ
Q03_WRITE
Output_Q4
Button_Q04
Output_Q4
Q04_READ
Q04_WRITE
Output_Q5
Button_Q05
Output_Q5
Q05_READ
Q05_WRITE
Output_Q6
Button_Q06
Output_Q6
Q06_READ
Q06_WRITE
Output_Q7
Button_Q07
Output_Q7
Q07_READ
Q07_WRITE
Output_Q8
Button_Q08
Output_Q8
Q08_READ
Q08_WRITE
Output_Q9
Button_Q09
Output_Q9
Q09_READ
Q09_WRITE
Output_Q10
Button_Q10
Output_Q10
Q10_READ
Q10_WRITE
Output_Q11
Button_Q11
Output_Q11
Q11_READ
Q11_WRITE
Output_Q12
Button_Q12
Output_Q12
Q12_READ
Q12_WRITE
Input_I1
logiBUS_IXA_BG_...
Button_I01
Input_I1
I1_WRITE
Input
Input_I2
Button_I02
Input_I2
I2_WRITE
Input_I3
Button_I03
Input_I3
I3_WRITE
Input_I4
Button_I04
Input_I4
I4_WRITE
Input_I5
Button_I05
Input_I5
I5_WRITE
Input_I6
Button_I06
Input_I6
I6_WRITE
Input_I7
Button_I07
Input_I7
I7_WRITE
Input_I8
Button_I08
Input_I8
I8_WRITE

• – ◦ • – ◦ • – ◦ •

Einleitung

`InputOutputTesterButton_DIDO_OPC_UA` ist das grundlegende Trainingsbeispiel für **8 digitale Eingänge und 12 digitale Ausgänge**, steuerbar sowohl über den ISOBUS-Virtual-Terminal als auch über OPC-UA (Web-Client). Es ist das rein digitale Gegenstück zum späteren PWM-Beispiel `InputOutputTesterButton_PWM_OPC_UA` — dessen 8 Eingänge unverändert aus diesem DIDO-Beispiel übernommen wurden, während die 12 rein digitalen Ausgänge dort durch 12 PWM-Kanäle ersetzt sind.

Die Übung ist ein reines Top-Level-Composite: Sie instanziiert 8 Eingangs-Bausteine, 12 Ausgangs-Bausteine und einen `SystemTickSender`, ohne selbst Logik zu enthalten — die eigentliche Verdrahtung (VT-Anzeige, OPC-UA-Publish/Subscribe, Rückkopplungs-Entkopplung) steckt vollständig in den wiederverwendbaren Sub-Bausteinen.

Verwendete Funktionsbausteine (FBs)

SubApp-Instanz	Typ	Zweck
`Input_I1` … `Input_I8`	`MyLib::sys::logiBUS_IXA_BG_OPC`	Digitaler Eingang mit VT-Statusanzeige (Grün/Weiß) und OPC-UA-Publish
`Output_Q1` … `Output_Q12`	`MyLib::sys::Button_IXA_TO_logiBUS_QXA_BG_OPC`	Digitaler Ausgang, bidirektional über VT-Taster UND OPC-UA schaltbar
`SystemTickSender`	`MyLib::sys::SystemTickSender`	Zyklischer Zähler für die VT-Statusanzeige (`OutputNumber_Tick`)

Sub-Baustein: logiBUS_IXA_BG_OPC (Eingänge)

- **Typ**: SubAppType (`MyLib::sys`)
- **Funktionsweise**: Liest einen physischen digitalen Eingang (`logiBUS_IXA`) und verzweigt das Adapter-Signal über `AX_SPLIT_2` in zwei Richtungen: an `GreenWhiteBackground1_AX` (VT-Hintergrundfarbe Grün/Weiß je nach Zustand) und an `AX_PUBLISH_1` (OPC-UA-Publish an den Web-Client). Reiner Einweg-Datenfluss physisch ⟶ VT/Web, keine Rückschreibmöglichkeit vom Web aus (Eingänge sind nicht extern schaltbar).

Sub-Baustein: Button_IXA_TO_logiBUS_QXA_BG_OPC (Ausgänge)

- **Typ**: SubAppType (`MyLib::sys`)
- **Funktionsweise**: Anders als bei den Eingängen kann ein Ausgang **von zwei Seiten** geschaltet werden — per VT-Taster (`Button_IXA`) oder per OPC-UA-Schreibzugriff (`AX_SUBSCRIBE_1`). Beide Quellen laufen über je einen `AX_RF_TRIG` (Flankenerkennung) auf ein gemeinsames `AX_SR`-Flipflop (Set/Reset), dessen Ausgang über `AX_SPLIT_3` dreifach verteilt wird: an den physischen Ausgang (`logiBUS_QXA`), an die VT-Statusfarbe (`GreenWhiteBackground1_AX`) und zurück an `AX_PUBLISH_1` (OPC-UA-Echo, damit der Web-Client den tatsächlichen Zustand sieht).
- **Bekannte Falle bei der OPC-UA-Rückkopplung**: Da `AX_PUBLISH_1` und `AX_SUBSCRIBE_1` denselben OPC-UA-Knoten bedienen, würde eine naive Verdrahtung einen Event-Storm erzeugen (jede eigene Veröffentlichung löst beim eigenen Subscribe erneut eine Indikation aus). Das `AX_SR`-Flipflop entkoppelt diese Rückkopplung, indem es nur echte Zustandswechsel (Set/Reset statt direkter Durchleitung) weiterreicht. Details siehe `NOTIZ_RSP_und_EventStorm.md` im Quell-Repository — dort auch der Befund, dass `SUBSCRIBE_1`s `RSP`-Event-Eingang aus FORTE-Quellcode-Sicht ein kompletter No-Op ist (nur aus der gemeinsamen `CCommFB`-Basisklasse geerbt, aber für den Service-Typ *Subscriber* nie wirksam).

OPC-UA-Adressraum

Im Gegensatz zum späteren, verschachtelten PWM-Adressraum verwendet DIDO flache, nach Signalart getrennte Knoten:

Node-Pfad	Node-ID	Bedeutung
`/Objects/DigitalInput/In`	`s=In`	Eingang n (n=1–8), nur Publish (Read-only für den Client)

`/Objects/DigitalOutput/Qnn`	`s=Qnn`	Ausgang nn (nn=01–12), Read (Subscribe) + Write (Publish/Echo)

Programmablauf und Verbindungen

Die Übung selbst enthält **keine Verbindungen** (`SubAppNetwork` besteht nur aus SubApp-Instanzen mit Parametern) — analog zum PWM-Beispiel ist die gesamte Logik in die Sub-Bausteine ausgelagert:

1. **8 Eingänge**: `Input_I1`…`Input_I8` lesen `Input_I1`…`Input_I8` und spiegeln sie per VT-Statusfarbe und OPC-UA-Publish (`I1_WRITE`…`I8_WRITE`).
2. **12 Ausgänge**: `Output_Q1`…`Output_Q12` verbinden je einen physischen Ausgang (`Output_Q1`…`Output_Q12`) mit VT-Taster, VT-Statusfarbe und bidirektionalem OPC-UA-Zugriff (`Q01_READ`…`Q12_READ` zum Schreiben vom Web, `Q01_WRITE`…`Q12_WRITE` als Echo).
3. **Zeitgeber**: `SystemTickSender` zählt zyklisch hoch und speist das VT-Zahlenfeld `OutputNumber_Tick` sowie den OPC-UA-Knoten `Tick_WRITE`.

Registrierung im Trainingssystem: Wie bei allen Übungen in diesem System kein eigenes `Application`-Element nötig — Auswahl per „Change Type“ im 4diac IDE auf den einen `Control`-Slot des `System`.

Lernziele

- Grundmuster für digitale Ein-/Ausgänge mit VT- **und** OPC-UA-Anbindung, bevor analoge Kanäle (PWM) hinzukommen.
- Unterschied zwischen reinem Publish (Eingänge, nur eine Datenquelle) und bidirektionalem Subscribe/Publish (Ausgänge, zwei konkurrierende Schreibquellen: VT-Taster und Web).
- Vermeidung von OPC-UA-Feedback-Loops bei gleichzeitigem Publish und Subscribe desselben Knotens — ein Muster, das in jedem bidirektional angebundenen Baustein dieses Trainingssystems wiederkehrt (vgl. `RampLimitFS_TO_logiBUS_QDA_PWM_OPC` im PWM-Beispiel, das dieselbe Herausforderung für den Kanal-Schalter löst).

Schwierigkeitsgrad: Einsteiger bis Mittel **Vorkenntnisse**: Grundlagen der logiBUS-Digital-I/O-Bausteine (`logiBUS_IXA`, `logiBUS_QXA`), OPC-UA-Adapter (`AX_SUBSCRIBE_1`/`AX_PUBLISH_1`).

Zusammenfassung

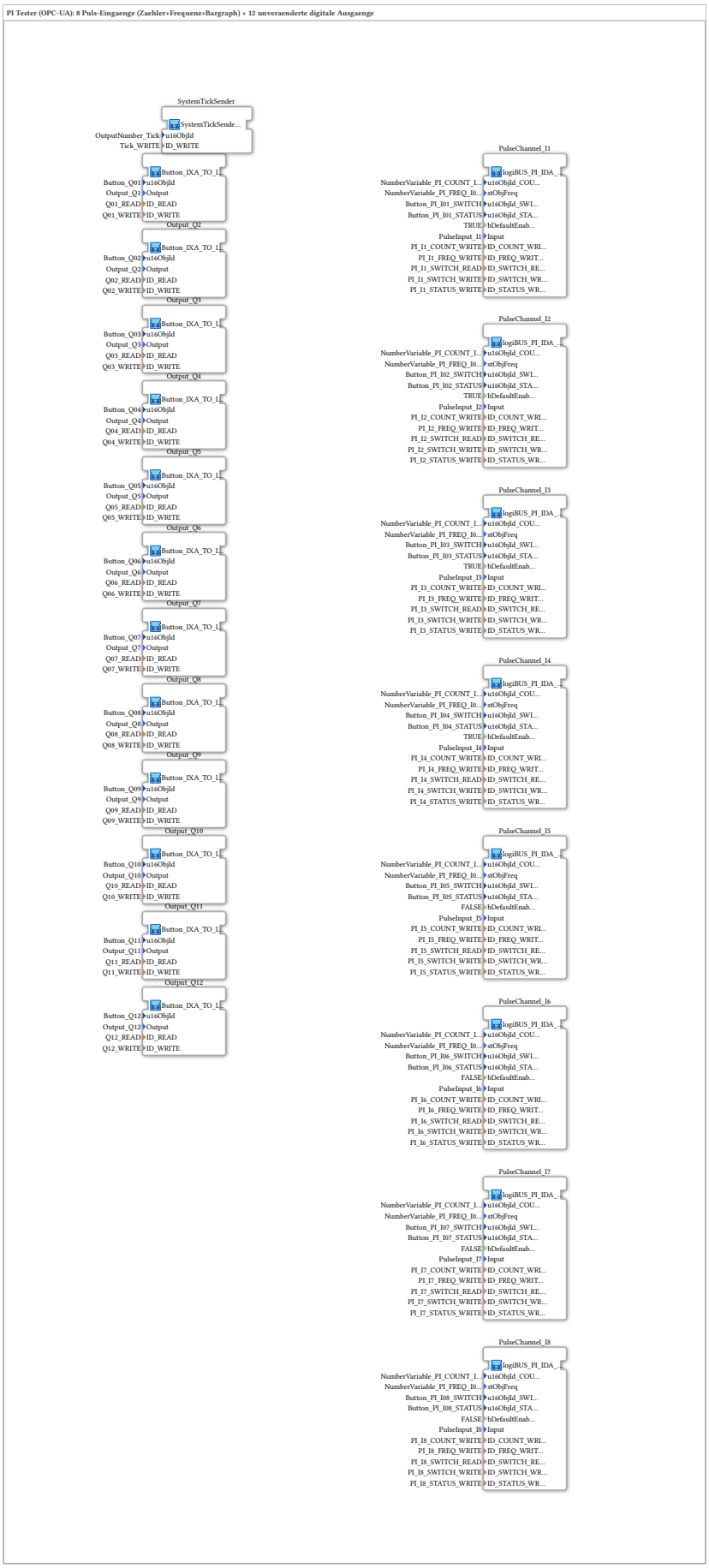
`InputOutputTesterButton_DIDO_OPC_UA` demonstriert das Grundmuster für digitale I/O mit paralleler VT- und OPC-UA-Bedienung: Eingänge als reiner Publish-Pfad, Ausgänge als bidirektionaler Subscribe/Publish-Pfad mit sauberer Entkopplung der Rückkopplungsschleife über ein Set/Reset-Flipflop. Dieses Muster — insbesondere die Feedback-Loop-Vermeidung — ist die Vorlage für alle späteren, komplexeren Trainingsbeispiele dieses Systems (DIDO → PWM → AI).

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

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

Button PI OPC UA

InputOutputTesterButton_PI_OPC_UA: PI Tester (OPC-UA)


PI Tester (OPC-UA): 8 Puls-Eingaenge (Zaehler+Frequenz+Bargraph) + 12 unveraenderte digitale Ausgaenge
SystemTickSender
SystemTickSende...
OutputNumber_Tick
Tick_WRITE
u16ObjId
ID_WRITE
Button_IXA_TO_1
Button_Q01
Output_Q1
Q01_READ
Q01_WRITE
Output_Q2
Button_Q02
Output_Q2
Q02_READ
Q02_WRITE
Output_Q3
Button_Q03
Output_Q3
Q03_READ
Q03_WRITE
Output_Q4
Button_Q04
Output_Q4
Q04_READ
Q04_WRITE
Output_Q5
Button_Q05
Output_Q5
Q05_READ
Q05_WRITE
Output_Q6
Button_Q06
Output_Q6
Q06_READ
Q06_WRITE
Output_Q7
Button_Q07
Output_Q7
Q07_READ
Q07_WRITE
Output_Q8
Button_Q08
Output_Q8
Q08_READ
Q08_WRITE
Output_Q9
Button_Q09
Output_Q9
Q09_READ
Q09_WRITE
Output_Q10
Button_Q10
Output_Q10
Q10_READ
Q10_WRITE
Output_Q11
Button_Q11
Output_Q11
Q11_READ
Q11_WRITE
Output_Q12
Button_Q12
Output_Q12
Q12_READ
Q12_WRITE
Output
ID_READ
PulseChannel_I1
PulseChannel_I2
PulseChannel_I3
PulseChannel_I4
PulseChannel_I5
PulseChannel_I6
PulseChannel_I7
PulseChannel_I8
logiBUS_PI_IDA_...
NumberVariable_PI_COUNT_I...
NumberVariable_PI_FREQ_10...
Button_PI_I01_SWITCH
Button_PI_I01_STATUS
PulseInput_I1
PI_I1_COUNT_WRITE
PI_I1_FREQ_WRITE
PI_I1_SWITCH_READ
PI_I1_SWITCH_WRITE
PI_I1_STATUS_WRITE
u16ObjId_COU...
stObjFreq
u16ObjId_SWI...
u16ObjId_STA...
bDefaultEnab...
Input
ID_COUNT_WRI...
ID_FREQ_WRIT...
ID_SWITCH_RE...
ID_SWITCH_WR...
ID_STATUS_WR...
PulseInput_I2
PI_I2_COUNT_WRITE
PI_I2_FREQ_WRITE
PI_I2_SWITCH_READ
PI_I2_SWITCH_WRITE
PI_I2_STATUS_WRITE
PulseInput_I3
PI_I3_COUNT_WRITE
PI_I3_FREQ_WRITE
PI_I3_SWITCH_READ
PI_I3_SWITCH_WRITE
PI_I3_STATUS_WRITE
PulseInput_I4
PI_I4_COUNT_WRITE
PI_I4_FREQ_WRITE
PI_I4_SWITCH_READ
PI_I4_SWITCH_WRITE
PI_I4_STATUS_WRITE
PulseInput_I5
PI_I5_COUNT_WRITE
PI_I5_FREQ_WRITE
PI_I5_SWITCH_READ
PI_I5_SWITCH_WRITE
PI_I5_STATUS_WRITE
PulseInput_I6
PI_I6_COUNT_WRITE
PI_I6_FREQ_WRITE
PI_I6_SWITCH_READ
PI_I6_SWITCH_WRITE
PI_I6_STATUS_WRITE
PulseInput_I7
PI_I7_COUNT_WRITE
PI_I7_FREQ_WRITE
PI_I7_SWITCH_READ
PI_I7_SWITCH_WRITE
PI_I7_STATUS_WRITE
PulseInput_I8
PI_I8_COUNT_WRITE
PI_I8_FREQ_WRITE
PI_I8_SWITCH_READ
PI_I8_SWITCH_WRITE
PI_I8_STATUS_WRITE

• – ◦ • – ◦ • – ◦ •

Einleitung

`InputOutputTesterButton_PI_OPC_UA` ist das Trainingsbeispiel für **8 Puls-Eingänge (Zähler + Frequenz + Bargraph)**, steuerbar sowohl über den ISOBUS-Virtual-Terminal als auch über OPC-UA. Die 12 digitalen Ausgänge sind unverändert aus `InputOutputTesterButton_DIDO_OPC_UA` übernommen („PI" = **P**ulse **I**nput, Impulseingang für Frequenzmessung — nicht zu verwechseln mit den analogen `AI`-Kanälen).

Jeder Pulskanal hat zusätzlich einen **eigenen Enable-Schalter mit Status-LED** — anders als bei DIDO/AI/AI_Calibrate ist hier also nicht nur der Messwert, sondern die Aktivierung des Kanals selbst bidirektional von VT und Web aus änderbar.

Verwendete Funktionsbausteine (FBs)

SubApp-Instanz	Typ	Zweck
`PulseChannel_I1` … `PulseChannel_I8`	`MyLib::sys::logiBUS_PI_IDA_OPC`	Pulseingang mit Zähler- und Frequenz-Anzeige (VT-Zahlenfelder + Bargraph), Enable-Schalter mit Status-LED, + OPC-UA
`Output_Q1` … `Output_Q12`	`MyLib::sys::Button_IXA_TO_logiBUS_QXA_BG_OPC`	Digitaler Ausgang, unverändert aus dem DIDO-Beispiel übernommen

Sub-Baustein: `logiBUS_PI_IDA_OPC` (Pulseingänge)

- **Typ**: SubAppType (`MyLib::sys`)
- **Funktionsweise**: Der physische Pulseingang (`logiBUS_PI`) liefert sowohl einen fortlaufenden Impulszähler (`COUNTVAR`, VT-Zahlenfeld) als auch die daraus abgeleitete Frequenz (`stObjFreq`, VT-Zahlenfeld mit Bargraph) — beide zusätzlich per OPC-UA veröffentlicht (`ID_COUNT_WRITE`, `ID_FREQ_WRITE`).
- **Enable-Schalter mit Status-Rückmeldung**: Jeder Kanal hat einen eigenen Enable-Schalter (`u16ObjId_SWITCH`, per VT-Taste **und** OPC-UA schaltbar über `ID_SWITCH_READ`/`ID_SWITCH_WRITE`) sowie eine Status-LED (`u16ObjId_STATUS`, per OPC-UA veröffentlicht über `ID_STATUS_WRITE`), die den tatsächlichen Aktivierungszustand des Kanals zurückmeldet — dasselbe Set/Reset-plus-Echo-Muster wie bei den bidirektional schaltbaren Ausgängen der anderen Beispiele, hier aber auf einen Eingangs-Freigabeschalter angewendet.
- **Default-Aktivierung pro Kanal**: `bDefaultEnabled` ist für die Kanäle 1–4 auf `TRUE`, für die Kanäle 5–8 auf `FALSE` voreingestellt — die zweite Kanalgruppe muss beim Start erst aktiv geschaltet werden, bevor sie zählt.

Sub-Baustein: Button_IXA_TO_logiBUS_QXA_BG_OPC (Ausgänge)

Unverändert aus dem DIDO-Beispiel — siehe dortige Beschreibung.

OPC-UA-Adressraum

Node-Pfad	Node-ID	Bedeutung
`/Objects/Pulse/In/COUNT`	`s=PI_In_COUNT`	Impulszähler Eingang n (n=1–8), nur Publish
`/Objects/Pulse/In/FREQ`	`s=PI_In_FREQ`	Frequenz Eingang n (n=1–8), nur Publish
`/Objects/Pulse/In/SWITCH`	`s=PI_In_SWITCH`	Enable-Schalter Eingang n, Read (Subscribe) + Write (Publish/Echo)
`/Objects/Pulse/In/STATUS`	`s=PI_In_STATUS`	Status-LED Eingang n, nur Publish
`/Objects/DigitalOutput/Qnn`	`s=Qnn`	Ausgang nn (nn=01–12), wie im DIDO-Beispiel

Programmablauf und Verbindungen

Die Übung selbst enthält **keine Verbindungen** (`SubAppNetwork` besteht nur aus SubApp-Instanzen mit Parametern):

1. **8 Pulskanäle**: `PulseChannel_I1`…`PulseChannel_I8` lesen `PulseInput_I1`…`PulseInput_I8`, zählen und messen deren Frequenz, veröffentlichen beides per OPC-UA und lassen sich einzeln per VT-Taste oder OPC-UA aktivieren/deaktivieren (Kanäle 1–4 initial aktiv, 5–8 initial inaktiv).
2. **12 digitale Ausgänge**: `Output_Q1`…`Output_Q12`, unverändert aus dem DIDO-Beispiel.

Registrierung im Trainingssystem: Wie bei allen Übungen in diesem System kein eigenes `Application`-Element nötig — Auswahl per „Change Type“ im 4diac IDE auf den einen `Control`-Slot des `System`.

Lernziele

- Impulszählung und Frequenzmessung als eigener Analogeingang-Typ, abzugrenzen von den spannungsbasierten `AI`-Kanälen.
- Bidirektional schaltbare Kanal-Freigabe mit Status-Rückmeldung (Enable/Status-Paar) als wiederverwendbares Muster für „Kanal ein-/ausschaltbar machen“ über VT und OPC-UA gleichzeitig.

Schwierigkeitsgrad: Mittel **Vorkenntnisse**: `InputOutputTesterButton_DIDO_OPC_UA` (Grundmuster VT+OPC-UA), `InputOutputTesterButton_AI_OPC_UA` (Vergleich zu einem einfacheren Analogeingang ohne Enable-Schalter).

Zusammenfassung

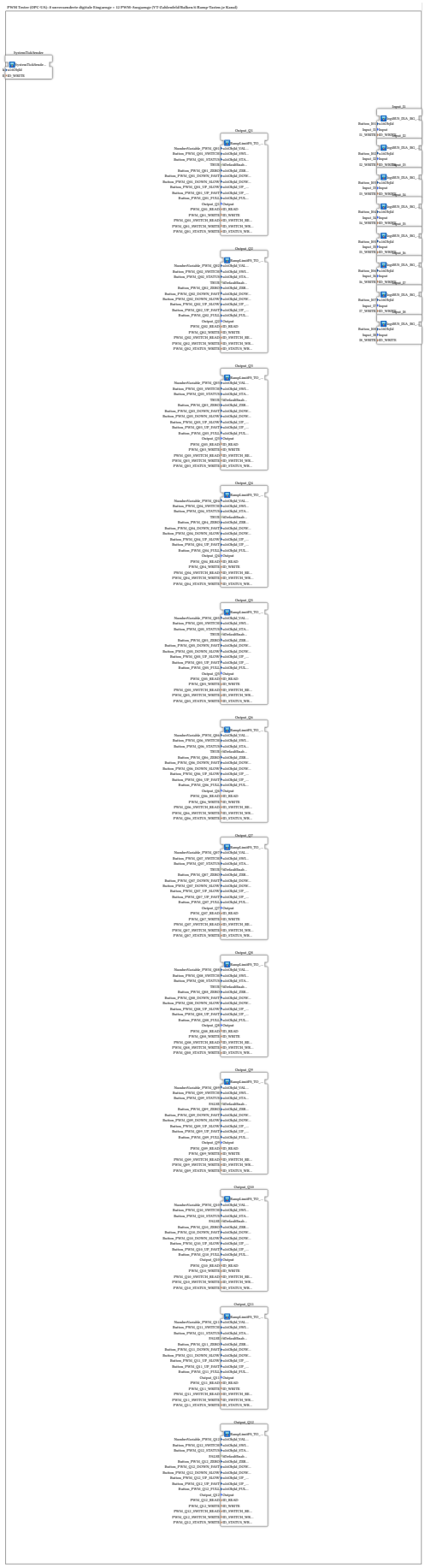
`InputOutputTesterButton_PI_OPC_UA` demonstriert Impulszählung mit Frequenzmessung und einem zusätzlichen, bidirektional schaltbaren Enable/Status-Paar je Kanal — eine Erweiterung des einfachen Publish-Musters (wie bei AI) um eine echte Rückkopplungslogik auf Eingangsseite.

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

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

Button PWM OPC UA

InputOutputTesterButton_PWM_OPC_UA: PWM Tester (OPC-UA)



• – ◦ • – ◦ • – ◦ •

Einleitung

`InputOutputTesterButton_PWM_OPC_UA` ist das PWM-Trainingsbeispiel zu **12 analogen Ausgängen (0–100 % Duty)**, steuerbar sowohl über den ISOBUS-Virtual-Terminal als auch über OPC-UA (Web-Client). Es ist das direkte Pendant zum bestehenden DI/DO-Beispiel `Meins::InputOutputTester::Button_DIDO_OPC_UA`: Die 8 digitalen Eingänge (`Input_I1`…`Input_I8`) bleiben unverändert, die früher rein digitalen Ausgänge werden hier durch **12 PWM-Kanäle** ersetzt, jeder mit VT-Zahlenfeld, Balkengrafik, 6 Ramp-Tasten, Kanal-Ein/Aus-Schalter und 3-Farben-Statusanzeige.

Da ein logiBUS-Controller nur **8 PWM-Kanäle gleichzeitig** physisch bedienen kann, sind beim Deployment die Kanäle 1–8 aktiv (`bDefaultEnabled=TRUE`) und die Kanäle 9–12 zunächst deaktiviert (`bDefaultEnabled=FALSE`); jeder Kanal lässt sich per Schalter unabhängig ein-/ausschalten.

Die Übung ist ein reines Top-Level-Composite: Sie instanziiert die 8 unveränderten Eingangs-Bausteine, 12× den PWM-Kanal-Baustein und einen `SystemTickSender`, ohne selbst Logik zu enthalten.

Verwendete Funktionsbausteine (FBs)

SubApp-Instanz	Typ	Zweck
`Input_I1` … `Input_I8`	`MyLib::sys::logiBUS_IXA_BG_OPC`	Unveränderte digitale Eingänge, je mit VT-Anzeige (`Button_I01..I08`) und OPC-UA-Publish (`I1_WRITE..I8_WRITE`)
`Output_Q1` … `Output_Q12`	`MyLib::sys::RampLimitFS_TO_logiBUS_QDA_PWM_OPC`	Je ein vollständiger PWM-Kanal (VT + physischer Ausgang + OPC-UA)
`SystemTickSender`	`MyLib::sys::SystemTickSender`	Zyklischer Zähler für die VT-Statusanzeige (`OutputNumber_Tick`)

Alle 12 `Output_Qxx`-Instanzen sind dieselbe wiederverwendbare Composite-SubApp `RampLimitFS_TO_logiBUS_QDA_PWM_OPC` (siehe Sub-Baustein unten) und unterscheiden sich nur durch ihre Parameter:

Sub-Baustein: RampLimitFS_TO_logiBUS_QDA_PWM_OPC

- **Typ**: SubAppType (`MyLib::sys`)
- **Instanziiert 12×** mit je eigenen Objekt-IDs für VT-Zahlenfeld/Balken (`u16ObjId_VALUEVAR`), 6 Ramp-Tasten (`u16ObjId_ZERO/DOWN_FAST/DOWN_SLOW/UP_SLOW/UP_FAST/FULL`), Kanal-Schalter (`u16ObjId_SWITCH`) und Status-Hintergrundfarbe (`u16ObjId_STATUS`), physischem PWM-Ausgang (`Output`, `logiBUS::io::DQ::logiBUS_DO_S`) sowie fünf OPC-UA-Adressen (`ID_READ`, `ID_WRITE`, `ID_SWITCH_READ`, `ID_SWITCH_WRITE`, `ID_STATUS_WRITE`).
- **Parameter `bDefaultEnabled`**: `TRUE` für Kanal 1–8, `FALSE` für Kanal 9–12 — legt den Enable-Zustand beim Deployment fest, da nur 8 PWM-Kanäle gleichzeitig physisch verfügbar sind.
- **Funktionsweise**: Verrechnet VT-Sollwert (Zahlenfeld/Ramp-Tasten) und Web-Sollwert (OPC-UA-Subscribe, Prozent REAL) auf denselben `RampLimitFS`-Rampenbaustein, gibt das Ergebnis als 13-Bit-PWM-Rohwert an den physischen `logiBUS_QD_PWM`-Ausgang und spiegelt Sollwert, Enable-Zustand und Ist-Status (`QO`) über OPC-UA-Publish zurück. Details siehe eigene Dokumentation des Bausteins.

OPC-UA-Adressraum

Für jeden Kanal `Qnn` (01–12) legt `SubStrings.gcf` einen eigenen Ordnerknoten unter `/Objects/PWM/Qnn/` an (FORTE erzeugt fehlende Zwischenordner automatisch als `FolderType`):

Node-Pfad	Node-ID	Richtung	Bedeutung
`/Objects/PWM/Qnn/VALUE`	`s=PWM_Qnn`	Read+Write	Sollwert Prozent (REAL 0.0–100.0)
`/Objects/PWM/Qnn/SWITCH`	`s=PWM_Qnn_SWITCH`	Read+Write	Kanal aktiviert/deaktiviert (BOOL)

/Objects/PWM/Qnn/ STATUS	s=PWM_Qnn_STATUS	Write (nur lesend für den Client)	Ist-Status logiBUS_QD_PWM.QO (BOOL)

Der Web-Client apixon-pwm-client bildet für jeden Kanal einen REAL-Slider mit Zahlenfeld (0–100 %) sowie die kombinierte 3-Farben-Statuslogik (Weiß=deaktiviert, Grün=aktiv+OK, Rot=aktiv+Störung) aus denselben zwei bereits publizierten Bits SWITCH/STATUS.

Programmablauf und Verbindungen

Die Übung selbst enthält **keine Verbindungen** (EventConnections/DataConnections sind leer) — sie besteht ausschließlich aus 21 parallel arbeitenden SubApp-Instanzen, deren Verdrahtung mit der physischen Hardware und den OPC-UA-Adressen über Parameter-Zuweisungen erfolgt:

1. **8 unveränderte Eingänge**: Input_I1…Input_I8 lesen die physischen Eingänge Input_I1…Input_I8 und spiegeln sie per VT-Anzeige (Button_I01…Button_I08) und OPC-UA-Publish (I1_WRITE…I8_WRITE).
2. **12 PWM-Kanäle**: Output_Q1…Output_Q12 verbinden je einen physischen PWM-Ausgang (Output_Q1…Output_Q12) mit einer eigenen Gruppe von VT-Objekten und OPC-UA-Adressen. Jede Instanz arbeitet unabhängig — es gibt keine Verbindungen zwischen den Kanälen.
3. **Zeitgeber**: SystemTickSender zählt zyklisch hoch und speist das VT-Zahlenfeld OutputNumber_Tick sowie den OPC-UA-Knoten Tick_WRITE — dient als „Lebenszeichen“ der Steuerung auf VT und Web-UI.

Registrierung im Trainingssystem: Kein eigenes Application-Element nötig. Das Trainingssystem besitzt genau einen Control-Slot pro System, der per „Change Type“ im 4diac IDE auf den gewünschten Übungstyp umgeschaltet wird — die PWM-Composite ist damit ein wählbares Ziel, ohne die Systemstruktur (test_AX.sys) zu ändern.

VT-Projekt: Workspace_PWM12 stellt die passenden ISO-Designer-Objekte bereit — eine Übersichtsseite plus 3 Datenmasken (DataMask_PWM1/2/3, 4 Kanäle je Seite), pro Kanal eine horizontale Zeile mit Label, Prozent-Zahlenfeld, Rohwert-Zahlenfeld, horizontalem Balkendiagramm und den 6 Ramp-Tasten in der Reihenfolge 0 -- - + ++ F.

Wertebereiche und Umrechnungsketten

Intern rechnet die Übung durchgängig in der SAE-J1939/ISO-11783-Fieldbus-Konvention (VALID_SIGNAL_W, 0–64255 statt einer frei erfundenen Skala):

- **VT/Web ↔ Fieldbus**: F_PWM_PERCENT_TO_RAW/F_PWM_RAW_TO_PERCENT (SubApp, MyLib::sys) rechnen zwischen Anteil 0.0–1.0 und Fieldbus-Rohwert 0–64255 um.
- **Prozent ↔ Anteil**: F_PERCENT_TO_FRACTION/F_FRACTION_TO_PERCENT (FBType, logiBUS::signalprocessing::fieldbus) rechnen zwischen Prozent 0–100 und Anteil 0.0–1.0 um — nötig, weil OPC-UA/Web in Prozent REAL kommuniziert, die internen Umrechnungsbausteine aber mit Anteil 0.0–1.0 arbeiten.
- **Fieldbus ↔ physischer PWM-Ausgang**: logiBUS_QD_PWM.OUT erwartet einen rohen 13-Bit-Wert (0–8191), daher rechnet der Kanal-Baustein zusätzlich ×8191 / 64255.

Lernziele

- Verständnis, wie ein Trainingsbeispiel mit vielen gleichartigen Kanälen durch **eine einzige wiederverwendbare Composite-SubApp** (statt kopierter Logik) aufgebaut wird — hier 12× RampLimitFS_TO_logiBUS_QDA_PWM_OPC mit unterschiedlicher Parametrierung.
- Umgang mit **analogen (PWM-)Ausgängen** im Gegensatz zu rein digitalen Ausgängen (Button_DIDO_OPC_UA), inklusive Sollwert-Ramping, Skalierungsketten und 13-Bit-Rohwertformat.
- Bidirektionale **OPC-UA-Synchronisation** eines analogen Sollwerts (VT ↔ Web, mit Mehrfachquellen-Mux über E_RS+F_SEL) sowie eines Kanal-Enable-Schalters.
- Grenzen der physischen Hardware (max. 8 gleichzeitige PWM-Kanäle) softwareseitig über einen individuell schaltbaren Enable-Zustand abbilden.

Schwierigkeitsgrad: Fortgeschritten **Vorkenntnisse**: Button_DIDO_OPC_UA (digitales Gegenstück), Uebung_094a (QI-basiertes Enable statt PERMIT), Grundlagen von RampLimitFS und OPC-UA-Adaptern (AR_SUBSCRIBE_1/AR_PUBLISH_1, AX_SUBSCRIBE_1/AX_PUBLISH_1).

Zusammenfassung

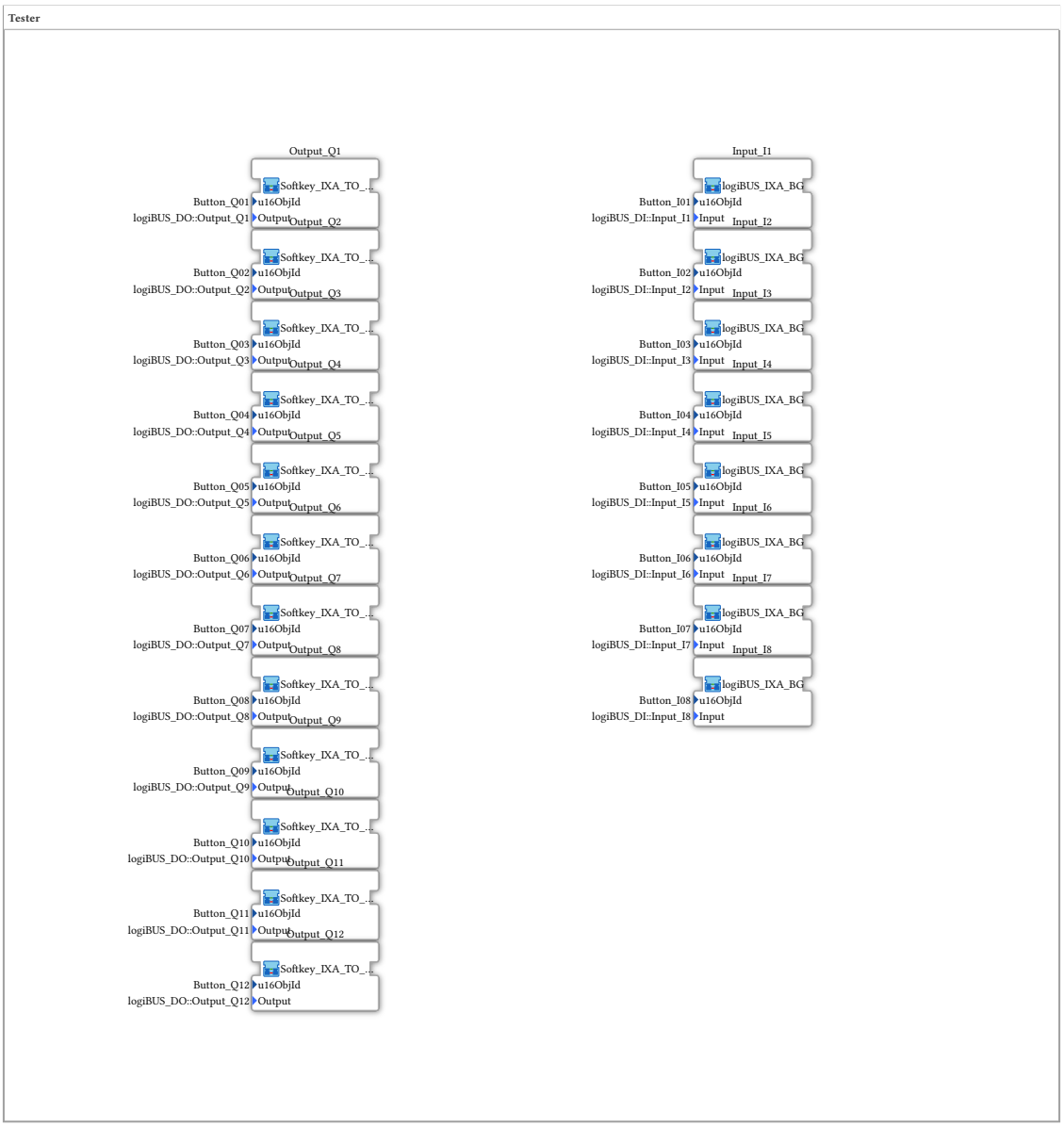
`InputOutputTesterButton_PWM_OPC_UA` demonstriert, wie sich ein komplettes 12-Kanal-PWM-Trainingsbeispiel mit VT- und Web-Bedienung allein durch mehrfache Instanziierung eines einzigen, gut gekapselten Kanal-Bausteins (`RampLimitFS_TO_logiBUS_QDA_PWM_OPC`) aufbauen lässt. Die eigentliche Komplexität — Sollwert-Ramping, Skalierung zwischen Prozent/Anteil/Fieldbus-Rohwert/13-Bit-PWM, Mehrfachquellen-Mux zwischen VT und Web sowie die 3-Farben-Statuslogik — steckt vollständig in diesem Sub-Baustein; die Top-Level-Übung selbst bleibt eine reine Parametrierungs- und Verdrahtungsliste.

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

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

SoftKey

InputOutputTesterSk: DIDO Tester (VT, SoftKey-Ausgänge)



• — ○ • — ○ • — ○ •

Einleitung

`InputOutputTesterSk` ist strukturell identisch zu `InputOutputTesterBt` (8 digitale Eingänge, 12 digitale Ausgänge, rein VT-basiert, keine OPC-UA-Anbindung) — der einzige Unterschied ist der Ausgangsbaustein: Die 12 Ausgänge werden hier über feste VT-**SoftKeys** (physische Tasten neben dem Bildschirm) geschaltet statt über in die DataMask eingebettete on-screen-Taster (`Button_IXA`).

Verwendete Funktionsbausteine (FBs)

SubApp-Instanz	Typ	Zweck
`Input_I1` … `Input_I8`	`MyLib::sys::logiBUS_IXA_BG`	Digitaler Eingang mit VT-Statusanzeige (Grün/Weiß) — unverändert aus `InputOutputTesterBt`
`Output_Q1` … `Output_Q12`	`MyLib::sys::Softkey_IXA_TO_logiBUS_QXA_BG`	Digitaler Ausgang, schaltbar über einen **VT-SoftKey** statt eines on-screen-Tasters

Sub-Baustein: `Softkey_IXA_TO_logiBUS_QXA_BG` (Ausgänge)

- **Typ**: SubAppType (`MyLib::sys`)
- **Funktionsweise**: Wie `Button_IXA_TO_logiBUS_QXA_BG` (AX_SR-Flipflop schaltet physischen Ausgang + VT-Statusfarbe), aber ausgelöst über eine `Softkey_IXA`-Verbindung (VT-SoftKeyMask-Taste) statt einer `Button_IXA`-Verbindung (VT-DataMask-Taster). Didaktisch relevanter Unterschied: SoftKeys sind maskenübergreifend fest positioniert (physische Tasten am VT-Gerät), während DataMask-Buttons Teil des jeweils angezeigten Bildschirminhalts sind.

Programmablauf und Verbindungen

Die Übung selbst enthält **keine Verbindungen** (`SubAppNetwork` besteht nur aus SubApp-Instanzen mit Parametern) — identisch zu `InputOutputTesterBt`:

1. **8 Eingänge**: `Input_I1`…`Input_I8` lesen `Input_I1`…`Input_I8` und spiegeln sie per VT-Statusfarbe.
2. **12 Ausgänge**: `Output_Q1`…`Output_Q12` verbinden je einen physischen Ausgang mit einem VT-SoftKey und VT-Statusfarbe.

Registrierung im Trainingssystem: Wie bei allen Übungen in diesem System kein eigenes `Application`-Element nötig — Auswahl per „Change Type“ im 4diac IDE auf den einen `Control`-Slot des `System`.

Lernziele

- Unterschied zwischen SoftKey- und DataMask-Button-Bedienung am selben Grundmuster (digitale I/O), ohne dass sich sonst etwas an der Logik ändert.
- SoftKeys als Alternative für Bedienelemente, die unabhängig vom aktuell angezeigten Maskeninhalt erreichbar sein müssen.

Schwierigkeitsgrad: Einsteiger **Vorkenntnisse**: `InputOutputTesterBt` (identisches Grundmuster mit DataMask-Buttons), Grundlagen der ISOBUS-VT-SoftKeyMasken.

Zusammenfassung

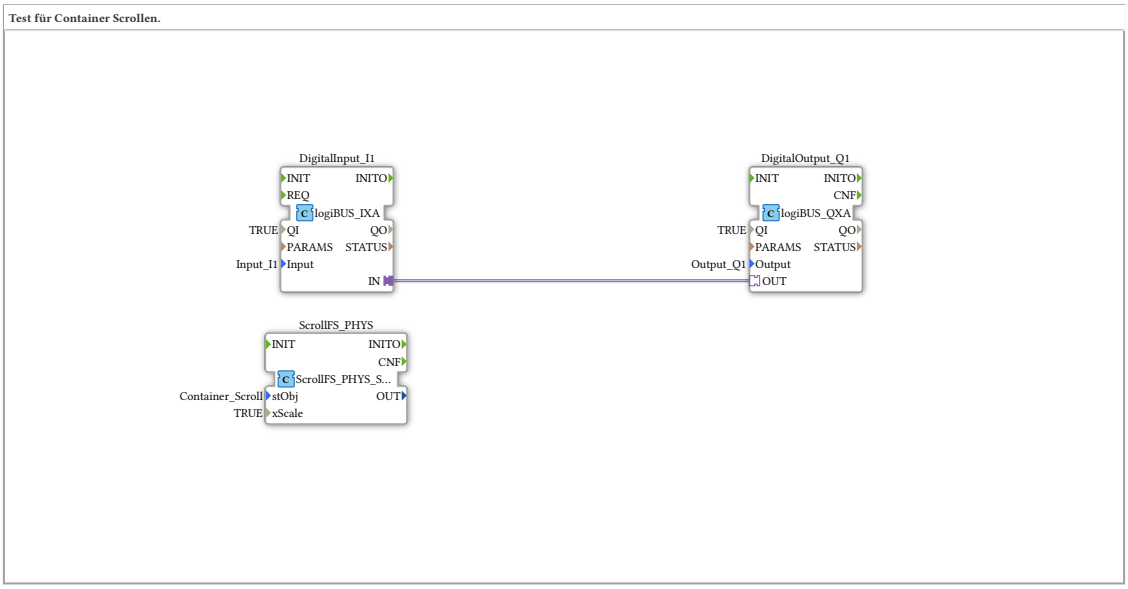
`InputOutputTesterSk` zeigt dasselbe digitale 8+12-I/O-Grundmuster wie `InputOutputTesterBt`, gesteuert über feste VT-SoftKeys statt on-screen-Buttons — eine reine Bedienkonzept-Variante ohne Änderung an der zugrundeliegenden FB-Logik.

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

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

2.1.1.10 Scroll

ScrollContainerTest: Test für Container-Scrollen



• – ◦ • – ◦ • – ◦ •

Einleitung

`ScrollContainerTest` ist ein Test für das Scrollen eines VT-Containers (Kommentar im Typ: „Test für Container Scrollen.“). Importiert wird die Pool-Konstante `Container_Scroll` aus `DefaultPool_Scroll`. Die SubApp enthält einen Scroll-Baustein für die Softkey-Bedienung sowie eine direkte Durchschaltung von Eingang I1 auf Ausgang Q1.

Verwendete Funktionsbausteine (FBs)

Instanz	Typ	Parameter	Erklärung
`DigitalInput_I1`	`logiBUS::io::DI::logiBUS_IXA`	`Input = Input_I1, QI = TRUE`	Liest den digitalen Eingang I1.
`DigitalOutput_Q1`	`logiBUS::io::DQ::logiBUS_QXA`	`Output = Output_Q1, QI = TRUE`	Schreibt den digitalen Ausgang Q1.
`ScrollFS_PHYS`	`logiBUS::UT::Q::ScrollFS_PHYS_Softkey`	`stObj = Container_Scroll, xScale = TRUE`	Scrollbare VT-Liste mit Softkey-Bedienung für den Container `Container_Scroll`.

Programmablauf und Verbindungen

Das Netzwerk enthält genau eine Verbindung:

1. `DigitalInput_I1.IN` ⟶ `DigitalOutput_Q1.OUT` (Adapter): der Eingang I1 wird direkt auf den Ausgang Q1 gelegt.

`ScrollFS_PHYS` ist im Netzwerk nicht verschaltet; er wird nur über seine Parameter (`stObj`, `xScale`) konfiguriert.

Zusammenfassung

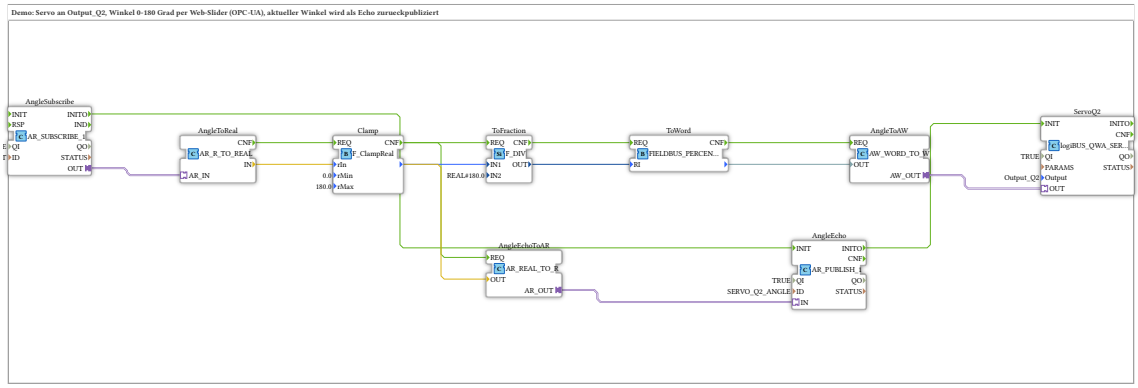
`ScrollContainerTest` ist ein kleiner Testaufbau: ein Scroll-Baustein für den Container `Container_Scroll` plus eine Eins-zu-eins-Durchschaltung von I1 nach Q1.

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

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

2.1.1.11 Servo Slider OPC

Servo_Slider_OPC: Servo per Web-Slider (OPC-UA)



• – ○ • – ○ • – ○ •

Einleitung

`Servo_Slider_OPC` ist eine Demo: Ein Servo an `Output_Q2` wird über einen Web-Slider per **OPC-UA** auf einen Winkel von **0 bis 180 Grad** gestellt; der tatsächlich verwendete (begrenzte) Winkel wird als Echo wieder per OPC-UA veröffentlicht. Die Schnittstelle der SubApp ist leer.

Die beiden OPC-UA-Verbindungen sind in `SubStrings.gcf` als WSTRING-Konstanten definiert:

Konstante	Wert	Richtung
`SERVO_Q2_ANGLE_SET`	`opc_ua[READ;/Objects/Servo/Q02/ANGLE_SET,1:s=SERVO_Q2_ANGLE_SET]`	Sollwinkel vom Slider lesen
`SERVO_Q2_ANGLE`	`opc_ua[WRITE;/Objects/Servo/Q02/ANGLE,1:s=SERVO_Q2_ANGLE]`	Echo des Winkels schreiben

Verwendete Funktionsbausteine (FBs)

Instanz	Typ	Parameter	Erklärung
`AngleSubscribe`	AR_SUBSCRIBE_1	`QI = TRUE, ID = SERVO_Q2_ANGLE_SET`	Liest den Sollwinkel per OPC-UA
`AngleToReal`	AR_R_TO_REAL	–	Wandelt den Adapterwert in einen `REAL`
`Clamp`	F_ClampReal	`rMin = 0.0, rMax = 180.0`	Begrenzt den Winkel auf 0…180
`ToFraction`	F_DIV	`IN2 = REAL#180.0`	Teilt den Winkel durch 180 (Anteil 0…1)
`ToWord`	FIELDBUS_PERCENT_TO_WORD	–	Wandelt den Anteil in ein Wort (Servo-Stellwert)
`AngleToAW`	AW_WORD_TO_W	–	Überträgt das Wort auf den WORD-Adapter des Servo-Ausgangs
`ServoQ2`	logiBUS_QWA_SERVO	`QI = TRUE, Output = Output_Q2`	Servo-Ausgang Q2
`AngleEchoToAR`	AR_REAL_TO_R	–	Wandelt den begrenzten Winkel in einen AR-Adapterwert
`AngleEcho`	AR_PUBLISH_1	`QI = TRUE, ID = SERVO_Q2_ANGLE`	Veröffentlicht den Winkel per OPC-UA

Programmablauf und Verbindungen

- **Initialisierung**: `AngleSubscribe.INITO` ⟶ `AngleEcho.INIT`, danach `AngleEcho.INITO` ⟶ `ServoQ2.INIT`.
- **Sollwert**: Das Subscribe-Ergebnis (`AngleSubscribe.OUT`) läuft über `AngleToReal` ⟶ `Clamp` ⟶ `ToFraction` ⟶ `ToWord` ⟶ `AngleToAW` auf `ServoQ2.OUT`. Jeder Schritt wird über sein `CNF`-Ereignis an den nächsten weitergegeben (`AngleToReal.CNF` ⟶ `Clamp.REQ` ⟶ `ToFraction.REQ` ⟶ `ToWord.REQ` ⟶ `AngleToAW.REQ`).
- **Echo**: Zusätzlich löst `Clamp.CNF` den Baustein `AngleEchoToAR` aus; dieser speist den begrenzten Winkel in `AngleEcho` ein, der ihn per OPC-UA veröffentlicht. Der Web-Client sieht also den tatsächlich ausgegebenen, auf 0…180 begrenzten Winkel.

Die Normierung (Winkel durch 180, dann Prozent-/Anteil-zu-Wort-Wandlung) bildet den Bereich 0…180 Grad auf den vollen Stellbereich des Servo-Ausgangs ab.

Zusammenfassung

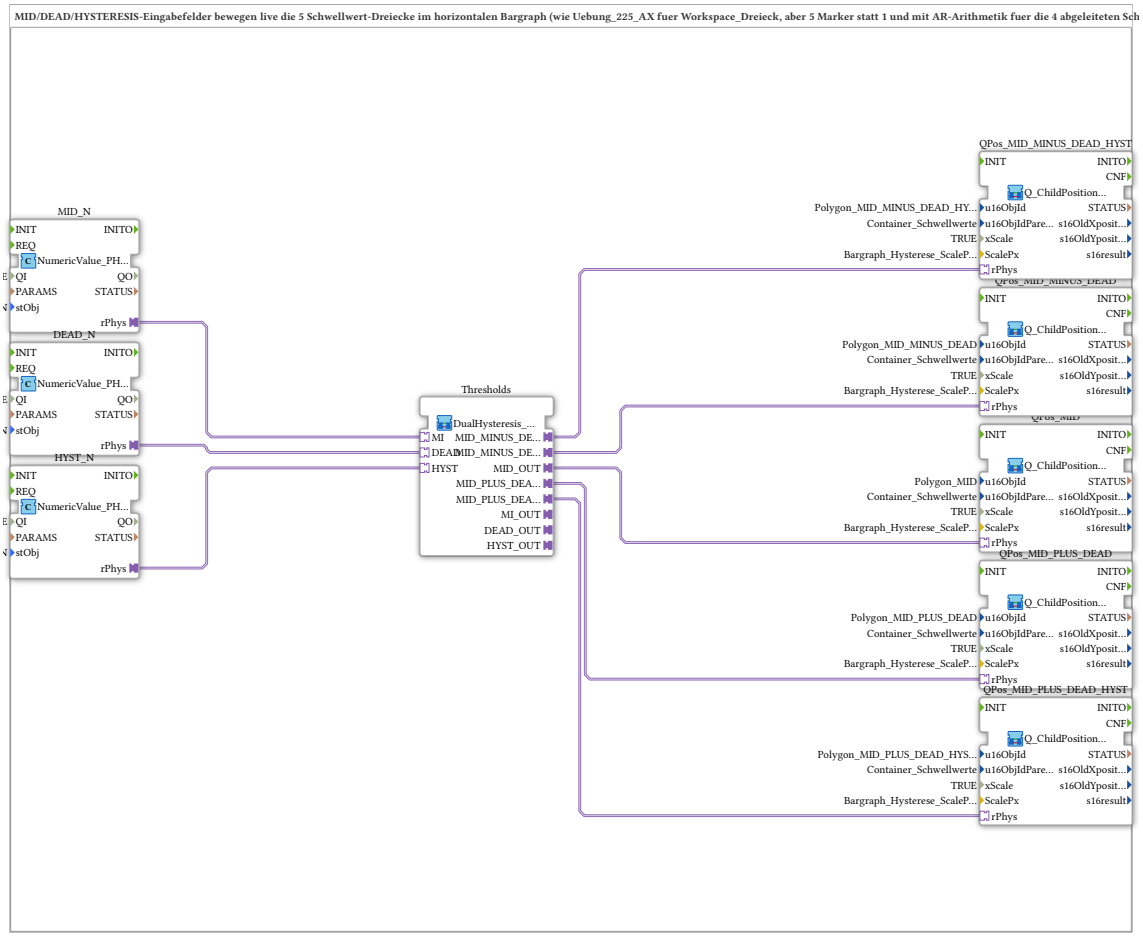
`Servo_Slider_OPC` zeigt eine Kette „OPC-UA lesen ⟶ begrenzen ⟶ normieren ⟶ Servo ausgeben“ mit einem parallelen Echo-Zweig, der den begrenzten Wert wieder per OPC-UA zurückmeldet.

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

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

2.1.1.12 Uebung HystereseHorizontal AX simple

Uebung_HystereseHorizontal_AX_simple: Hysterese-Schwellwerte im horizontalen Bargraph



• – ○ • – ○ • – ○ •

Einleitung

`Uebung_HystereseHorizontal_AX_simple` ist eine Übung, bei der die drei VT-Eingabefelder `MID`, `DEAD` und `HYSTERESIS` live fünf Schwellwert-Dreiecke im **horizontalen** Bargraph verschieben. Laut Kommentar im Typ entspricht sie `Uebung_225_AX` (Workspace_Dreieck), hat aber fünf Marker statt einem und nutzt AR-Arithmetik für die vier abgeleiteten Schwellen. Die SubApp hat keine eigene Schnittstelle und keine Ereignisverbindungen.

Verwendete Funktionsbausteine (FBs)

Instanz	Typ	Parameter	Erklärung
`MID_N`	`MyLibs::UT::io::NumericValue::NumericValue_TRUESA`	`QI := TRUE, stObj := InputNumber_MID_N`	Eingabefeld MID des VT; liefert den physikalischen Wert am Adapter `rPhys`.
`DEAD_N`	`MyLibs::UT::io::NumericValue::NumericValue_TRUESA`	`QI := TRUE, stObj := InputNumber_DEAD_N`	Eingabefeld DEAD des VT; liefert den physikalischen Wert am Adapter `rPhys`.
`HYST_N`	`MyLibs::UT::io::NumericValue::NumericValue_TRUESA`	`QI := TRUE, stObj := InputNumber_HYSTERESIS_N`	Eingabefeld HYSTERESIS des VT; liefert den physikalischen Wert am Adapter `rPhys`.
`Thresholds`	`MyLib::sys::DualHysteresis_Thresholds_AR`	–	Berechnet aus `MI`, `DEAD` und `HYST` die fünf Schwellwerte.
`QPos_MID`	`MyLibs::sys::Q_ChildPosition_AR_Bargraph_LeftToRight`	`u16ObjId := Polygon_MID, u16ObjIdParent := Container_Schwellwerte, xScale := TRUE, ScalePx := Bargraph_Hysterese_ScalePx`	Positioniert das Dreieck `Polygon_MID` für den Schwellwert Mitte (`MID`).
`QPos_MID_PLUS_DEAD`	`MyLibs::sys::Q_ChildPosition_AR_Bargraph_LeftToRight`	`u16ObjId := Polygon_MID_PLUS_DEAD, u16ObjIdParent := Container_Schwellwerte, xScale := TRUE, ScalePx := Bargraph_Hysterese_ScalePx`	Positioniert das Dreieck `Polygon_MID_PLUS_DEAD` für den Schwellwert `MID + DEAD`.
`QPos_MID_MINUS_DEAD`	`MyLibs::sys::Q_ChildPosition_AR_Bargraph_LeftToRight`	`u16ObjId := Polygon_MID_MINUS_DEAD, u16ObjIdParent := Container_Schwellwerte, xScale := TRUE, ScalePx := Bargraph_Hysterese_ScalePx`	Positioniert das Dreieck `Polygon_MID_MINUS_DEAD` für den Schwellwert `MID - DEAD`.
`QPos_MID_PLUS_DEAD_HYST`	`MyLibs::sys::Q_ChildPosition_AR_Bargraph_LeftToRight`	`u16ObjId := Polygon_MID_PLUS_DEAD_HYST, u16ObjIdParent := Container_Schwellwerte, xScale := TRUE, ScalePx := Bargraph_Hysterese_ScalePx`	Positioniert das Dreieck `Polygon_MID_PLUS_DEAD_HYST` für den Schwellwert `MID + DEAD + HYST`.
`QPos_MID_MINUS_DEAD_HYST`	`MyLibs::sys::Q_ChildPosition_AR_Bargraph_LeftToRight`	`u16ObjId := Polygon_MID_MINUS_DEAD_HYST, u16ObjIdParent := Container_Schwellwerte, xScale := TRUE, ScalePx := Bargraph_Hysterese_ScalePx`	Positioniert das Dreieck `Polygon_MID_MINUS_DEAD_HYST` für den Schwellwert `MID - DEAD - HYST`.

Programmablauf und Verbindungen

Alle Verbindungen sind Adapterverbindungen (AR, `REAL`-Werte):

1. `MID_N.rPhys` ⟶ `Thresholds.MI`
2. `DEAD_N.rPhys` ⟶ `Thresholds.DEAD`
3. `HYST_N.rPhys` ⟶ `Thresholds.HYST`
4. `Thresholds.MID_OUT` ⟶ `QPos_MID.rPhys`

5. `Thresholds.MID_PLUS_DEAD` ⟶ `QPos_MID_PLUS_DEAD.rPhys`
6. `Thresholds.MID_MINUS_DEAD` ⟶ `QPos_MID_MINUS_DEAD.rPhys`
7. `Thresholds.MID_PLUS_DEAD_HYST` ⟶ `QPos_MID_PLUS_DEAD_HYST.rPhys`
8. `Thresholds.MID_MINUS_DEAD_HYST` ⟶ `QPos_MID_MINUS_DEAD_HYST.rPhys`

Die Schwellen berechnet `Thresholds` aus Mitte, Totzone und Hysterese; `MID_PLUS_DEAD_HYST` ist `MI + (DEAD + HYST)`, `MID_MINUS_DEAD_HYST` ist `MI - (DEAD + HYST)`. `Q_ChildPosition_AR_Bargraph_LeftToRight` positioniert das jeweilige Kind-Objekt in X-Richtung (X = Wert × `ScalePx`, Y = 0); ein größerer Wert schiebt das Dreieck nach rechts.

Zusammenfassung

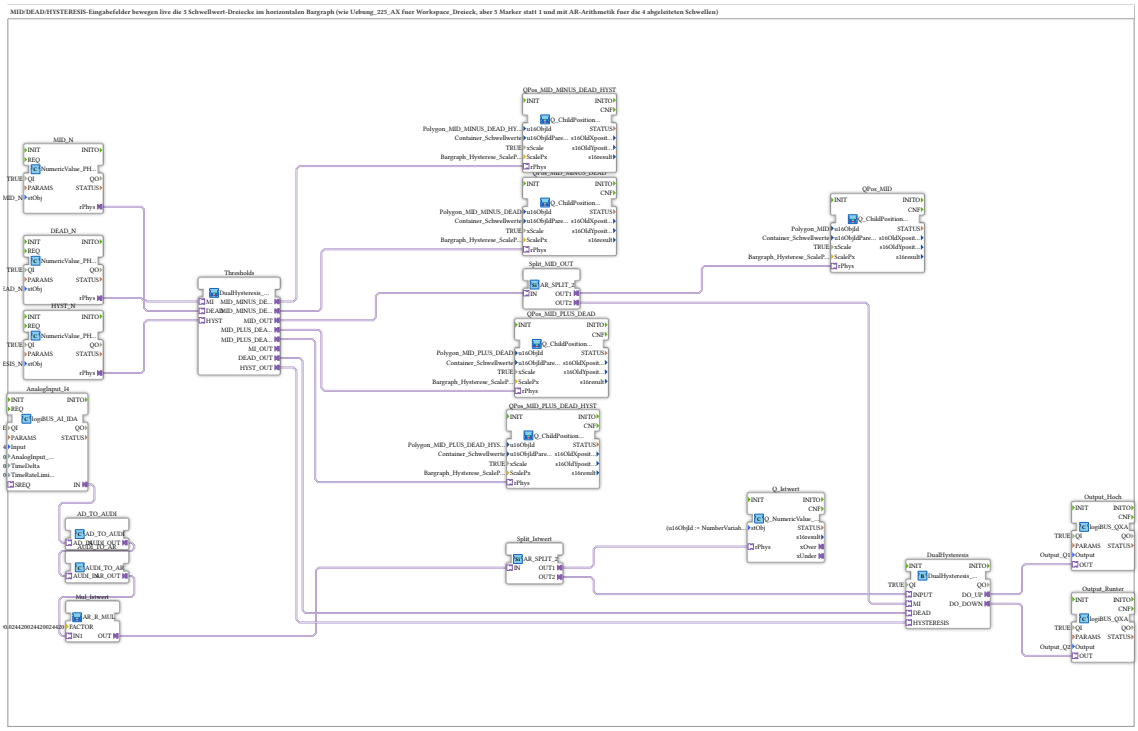
`Uebung_HystereseHorizontal_AX_simple` verbindet drei Eingabefelder über die Schwellwert-Berechnung mit fünf Bargraph-Positionsbausteinen, sodass die Marker live den eingegebenen Werten folgen.

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

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

2.1.1.13 Uebung HystereseHorizontal AX withAnalog

Uebung_HystereseHorizontal_AX_withAnalog: Hysterese-Schwellen im horizontaler Bargraph



• — ○ • — ○ • — ○ •

Einleitung

`Uebung_HystereseHorizontal_AX_withAnalog` zeigt eine **Zwei-Wege-Hysterese** (Heben/Senken) mit analogem Istwert und Bedienung über das ISOBUS-Terminal: Über die Eingabefelder **MID**, **DEAD** und **HYSTERESIS** werden Mittelwert, Totzone und Hysterese vorgegeben. Daraus werden fünf Schwellwerte berechnet, die als fünf Dreiecke (Polygone) live im horizontalen Bargraph (von links nach rechts) mitwandern. Der Istwert vom Analogeingang `AnalogInput_I4` wird als Zahl angezeigt und steuert über die Hysterese die Ausgänge `Output_Q1` (hoch) und `Output_Q2` (runter).

Die Schnittstelle der SubApp ist leer. Die Pool-Objekte stammen aus `Uebungen::const::UT::HystereseHorizontal` (`DefaultPool`, `DefaultPool_Numeric`, `DefaultPool_Bargraph`).

Verwendete Funktionsbausteine (FBs)

Eingabefelder und Istwert

Instanz	Typ	Parameter	Erklärung
MID_N	NumericValue_PHYSA	stObj = InputNumber_MID_N	Eingabefeld Mittelwert (MID), liefert rPhys
DEAD_N	NumericValue_PHYSA	stObj = InputNumber_DEAD_N	Eingabefeld Totzone (DEAD)
HYST_N	NumericValue_PHYSA	stObj = InputNumber_HYSTERESIS_N	Eingabefeld Hysterese
AnalogInput_I4	logiBUS_AI_IDA	Input = AnalogInput_I4, AnalogInput_hysteresis = DWORD#0, TimeDelta = DWORD#250, TimeRateLimit = DWORD#100	Analogeingang I4
AD_TO_AUDI	AD_TO_AUDI	–	Adapter-Konvertierung AD ⟶ AUDI
AUDI_TO_AR	AUDI_TO_AR	–	Adapter-Konvertierung AUDI ⟶ AR (REAL)
Mul_Istwert	AR_R_MUL	FACTOR = REAL#0.024420024420024420	Skaliert den Rohwert (Faktor = 100/4095)
Split_Istwert	AR_SPLIT_2	–	Verteilt den Istwert auf Anzeige und Hysterese
Q_Istwert	Q_NumericValue_PHYSA	stObj = (u16ObjId := NumberVariable_Istwert, r32Scale := 1.0, i32Offset := 0, u8Decimals := 0)	Zeigt den Istwert im Terminal an

Schwellwerte und Hysterese

Instanz	Typ	Parameter	Erklärung
Thresholds	DualHysteresis_Thresholds_AR	–	Berechnet aus MID, DEAD und HYST die fünf Schwellen (MID, MID±DEAD, MID±(DEAD+HYST))
Split_MID_OUT	AR_SPLIT_2	–	Verteilt MID_OUT auf den Mittelwert-Marker und die Hysterese
DualHysteresis	DualHysteresis_AR_AX	QI = TRUE	Zwei-Wege-Hysterese mit INPUT, MI, DEAD, HYSTERESIS; Ausgänge DO_UP, DO_DOWN
Output_Hoch	logiBUS_QXA	QI = TRUE, Output = Output_Q1	Ausgang „hoch“
Output_Runter	logiBUS_QXA	QI = TRUE, Output = Output_Q2	Ausgang „runter“

Schwellwert-Dreiecke im Bargraph

Instanz	Typ	Parameter (u16ObjId)	Erklärung

QPos_MID	Q_ChildPosition_AR_Bargraph_LeftToRight	Polygon_MID	Marker für MID
QPos_MID_PLUS_DEAD	Q_ChildPosition_AR_Bargraph_LeftToRight	Polygon_MID_PLUS_DEAD	Marker für MID + DEAD
QPos_MID_MINUS_DEAD	Q_ChildPosition_AR_Bargraph_LeftToRight	Polygon_MID_MINUS_DEAD	Marker für MID − DEAD
QPos_MID_PLUS_DEAD_HYST	Q_ChildPosition_AR_Bargraph_LeftToRight	Polygon_MID_PLUS_DEAD_HYST	Marker für MID + (DEAD + HYST)
QPos_MID_MINUS_DEAD_HYST	Q_ChildPosition_AR_Bargraph_LeftToRight	Polygon_MID_MINUS_DEAD_HYST	Marker für MID − (DEAD + HYST)

Gemeinsame Parameter aller fünf `QPos_*`-Instanzen: `u16ObjId`, `u16ObjIdParent` = `Container_Schwellwerte`, `xScale` = `TRUE`, `ScalePx` = `Bargraph_Hysterese_ScalePx`.

Programmablauf und Verbindungen

1. **Vorgabe**: `MID_N.rPhys`, `DEAD_N.rPhys` und `HYST_N.rPhys` gehen auf `Thresholds.MI`, `.DEAD` und `.HYST`.
2. **Schwellen → Anzeige**: Die vier abgeleiteten Schwellen (`MID_PLUS_DEAD`, `MID_MINUS_DEAD`, `MID_PLUS_DEAD_HYST`, `MID_MINUS_DEAD_HYST`) gehen jeweils auf `rPhys` des zugehörigen `QPos_*`; `MID_OUT` läuft über `Split_MID_OUT` (`OUT1` → `QPos_MID.rPhys`, `OUT2` → `DualHysteresis.MI`). Verschiebt der Bediener ein Eingabefeld, wandern die Dreiecke live mit.
3. **Hysterese-Parameter**: `Thresholds.DEAD_OUT` → `DualHysteresis.DEAD`, `Thresholds.HYST_OUT` → `DualHysteresis.HYSTERESIS`.
4. **Istwert**: `AnalogInput_I4.IN` → `AD_TO_AUDI` → `AUDI_TO_AR` → `Mul_Istwert` → `Split_Istwert`; `OUT1` → `Q_Istwert.rPhys` (Anzeige), `OUT2` → `DualHysteresis.INPUT`.
5. **Ausgänge**: `DualHysteresis.DO_UP` → `Output_Hoch` (Q1), `DualHysteresis.DO_DOWN` → `Output_Runter` (Q2).

Versionshistorie laut Quelle: V1.1 – AR_TO_AI, Zero und QPos in die SubApp `Q_ChildPosition_AR_*` ausgelagert; V1.2 – Schwellwertberechnung in `DualHysteresis_Thresholds_AR` ausgelagert.

Zusammenfassung

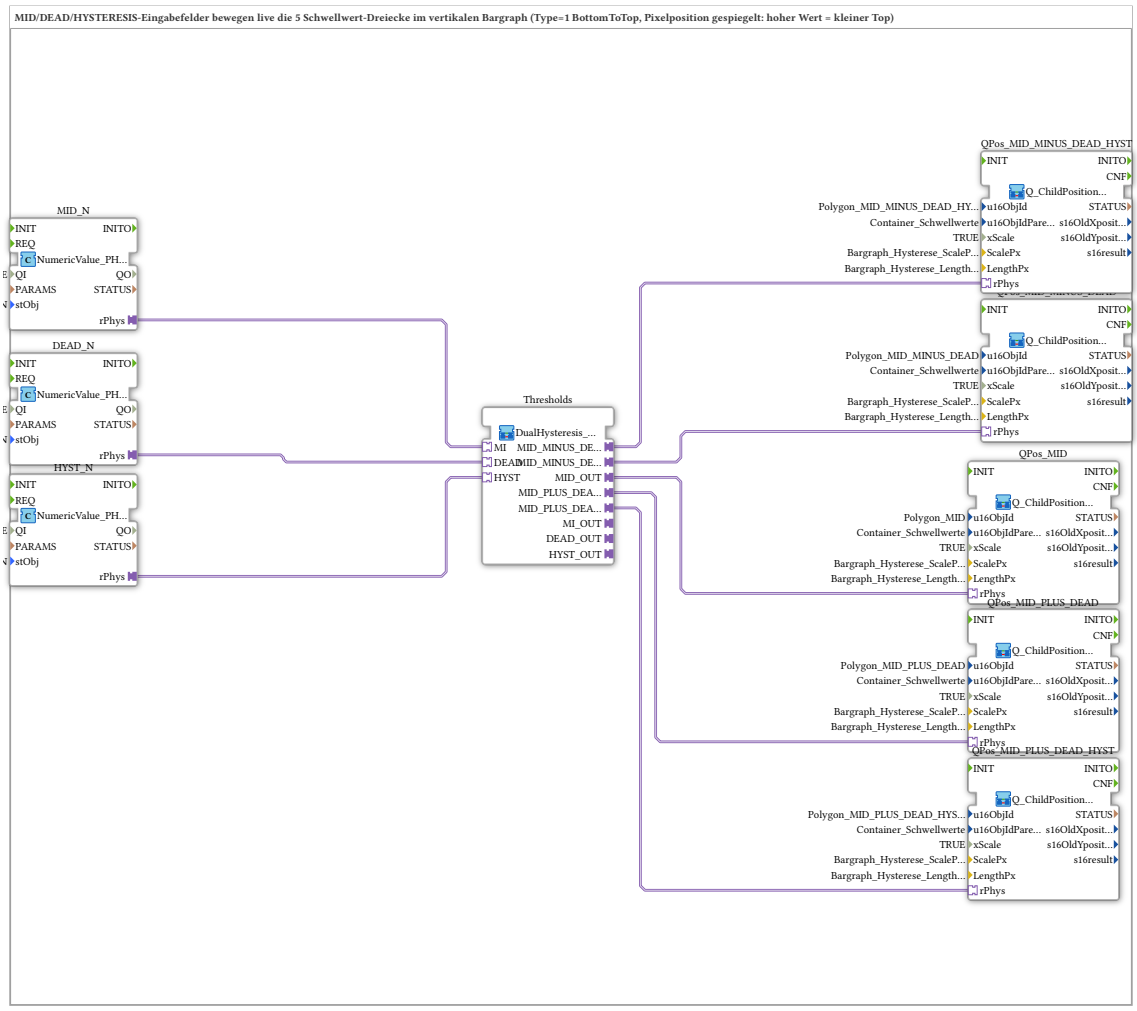
`Uebung_HystereseHorizontal_AX_withAnalog` verbindet drei Eingabefelder, eine Analogmessung, eine Zwei-Wege-Hysterese und die Live-Positionierung von fünf Schwellwert-Dreiecken in einem horizontaler Bargraph – komplett über Adapter-Verbindungen, ohne Ereignisverdrahtung.

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

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

2.1.1.14 Uebung HystereseVertikal AX simple

Uebung_HystereseVertikal_AX_simple: Hysterese-Schwellwerte im vertikalen Bargraph



• – ○ • – ○ • – ○ •

Einleitung

`Uebung_HystereseVertikal_AX_simple` ist eine Übung, bei der die drei VT-Eingabefelder `MID`, `DEAD` und `HYSTERESIS` live fünf Schwellwert-Dreiecke im **vertikalen** Bargraph (Type 1, BottomToTop) verschieben. Laut Kommentar im Typ ist die Pixelposition gespiegelt: ein hoher Wert ergibt ein kleines Top. Die SubApp hat keine eigene Schnittstelle und keine Ereignisverbindungen.

Verwendete Funktionsbausteine (FBs)

Instanz	Typ	Parameter	Erklärung
`MID_N`	`fbAdvUs::UT::io::NumericValue::NumericValue_PHYSA`	`QI := TRUE, stObj := InputNumber_MID_N`	Eingabefeld MID des VT; liefert den physikalischen Wert am Adapter `rPhys`.
`DEAD_N`	`fbAdvUs::UT::io::NumericValue::NumericValue_PHYSA`	`QI := TRUE, stObj := InputNumber_DEAD_N`	Eingabefeld DEAD des VT; liefert den physikalischen Wert am Adapter `rPhys`.
`HYST_N`	`fbAdvUs::UT::io::NumericValue::NumericValue_PHYSA`	`QI := TRUE, stObj := InputNumber_HYSTERESIS_N`	Eingabefeld HYSTERESIS des VT; liefert den

			physikalischen Wert am Adapter `rPhys`.
`Thresholds`	`MyLib::sys::DualHysteresis_Thresholds_AR`	-	Berechnet aus `MI`, `DEAD` und `HYST` die fünf Schwellwerte.
`QPos_MID`	`MyLib::sys::Q_ChildPosition_AR_Bargraph_BottomToTop`	`u16ObjId = Polygon_MID, u16ObjIdParent = Container_Schwellwerte, xScale = TRUE, ScalePx = Bargraph_Hysterese_ScalePx, LengthPx = Bargraph_Hysterese_LengthPx`	Positioniert das Dreieck `Polygon_MID` für den Schwellwert Mitte (`MID`).
`QPos_MID_PLUS_DEAD`	`MyLib::sys::Q_ChildPosition_AR_Bargraph_BottomToTop`	`u16ObjId = Polygon_MID_PLUS_DEAD, u16ObjIdParent = Container_Schwellwerte, xScale = TRUE, ScalePx = Bargraph_Hysterese_ScalePx, LengthPx = Bargraph_Hysterese_LengthPx`	Positioniert das Dreieck `Polygon_MID_PLUS_DEAD` für den Schwellwert `MID + DEAD`.
`QPos_MID_MINUS_DEAD`	`MyLib::sys::Q_ChildPosition_AR_Bargraph_BottomToTop`	`u16ObjId = Polygon_MID_MINUS_DEAD, u16ObjIdParent = Container_Schwellwerte, xScale = TRUE, ScalePx = Bargraph_Hysterese_ScalePx, LengthPx = Bargraph_Hysterese_LengthPx`	Positioniert das Dreieck `Polygon_MID_MINUS_DEAD` für den Schwellwert `MID - DEAD`.
`QPos_MID_PLUS_DEAD_HYST`	`MyLib::sys::Q_ChildPosition_AR_Bargraph_BottomToTop`	`u16ObjId = Polygon_MID_PLUS_DEAD_HYST, u16ObjIdParent = Container_Schwellwerte, xScale = TRUE, ScalePx = Bargraph_Hysterese_ScalePx, LengthPx = Bargraph_Hysterese_LengthPx`	Positioniert das Dreieck `Polygon_MID_PLUS_DEAD_HYST` für den Schwellwert `MID + DEAD + HYST`.
`QPos_MID_MINUS_DEAD_HYST`	`MyLib::sys::Q_ChildPosition_AR_Bargraph_BottomToTop`	`u16ObjId = Polygon_MID_MINUS_DEAD_HYST, u16ObjIdParent = Container_Schwellwerte, xScale = TRUE, ScalePx = Bargraph_Hysterese_ScalePx, LengthPx = Bargraph_Hysterese_LengthPx`	Positioniert das Dreieck `Polygon_MID_MINUS_DEAD_HYST` für den Schwellwert `MID - DEAD - HYST`.

Programmablauf und Verbindungen

Alle Verbindungen sind Adapterverbindungen (AR, `REAL`-Werte):

1. `MID_N.rPhys` ⟶ `Thresholds.MI`
2. `DEAD_N.rPhys` ⟶ `Thresholds.DEAD`
3. `HYST_N.rPhys` ⟶ `Thresholds.HYST`
4. `Thresholds.MID_OUT` ⟶ `QPos_MID.rPhys`
5. `Thresholds.MID_PLUS_DEAD` ⟶ `QPos_MID_PLUS_DEAD.rPhys`
6. `Thresholds.MID_MINUS_DEAD` ⟶ `QPos_MID_MINUS_DEAD.rPhys`
7. `Thresholds.MID_PLUS_DEAD_HYST` ⟶ `QPos_MID_PLUS_DEAD_HYST.rPhys`
8. `Thresholds.MID_MINUS_DEAD_HYST` ⟶ `QPos_MID_MINUS_DEAD_HYST.rPhys`

Die Schwellen berechnet `Thresholds` aus Mitte, Totzone und Hysterese; `MID_PLUS_DEAD_HYST` ist `MI + (DEAD + HYST)`, `MID_MINUS_DEAD_HYST` ist `MI - (DEAD + HYST)`. `Q_ChildPosition_AR_Bargraph_BottomToTop` skaliert den Wert mit `ScalePx` und spiegelt ihn an `LengthPx` (Y = `LengthPx` − Wert × `ScalePx`, X = 0); ein größerer Wert schiebt das Dreieck nach oben.

Zusammenfassung

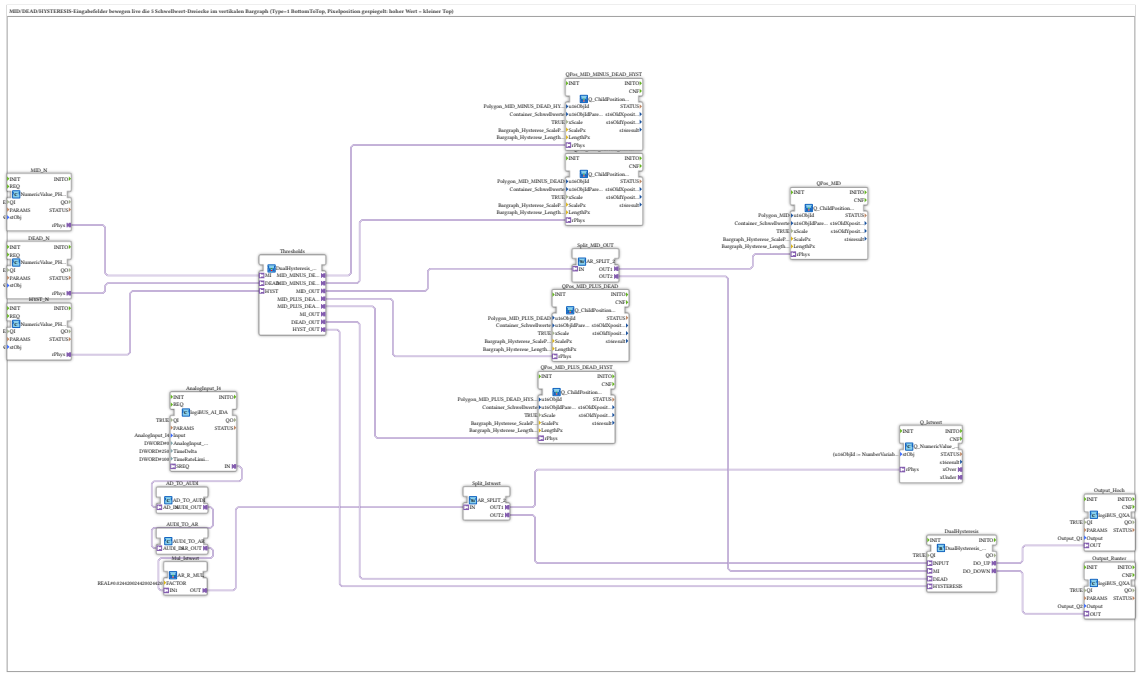
`Uebung_HystereseVertikal_AX_simple` verbindet drei Eingabefelder über die Schwellwert-Berechnung mit fünf Bargraph-Positionsbausteinen, sodass die Marker live den eingegebenen Werten folgen.

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

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

2.1.1.15 Uebung HystereseVertikal AX withAnalog

Uebung_HystereseVertikal_AX_withAnalog: Hysterese-Schwellen im vertikaler Bargraph



• – ○ • – ○ • – ○ •

Einleitung

`Uebung_HystereseVertikal_AX_withAnalog` zeigt eine **Zwei-Wege-Hysterese** (Heben/Senken) mit analogem Istwert und Bedienung über das ISOBUS-Terminal: Über die Eingabefelder **MID**, **DEAD** und **HYSTERESIS** werden Mittelwert, Totzone und Hysterese vorgegeben. Daraus werden fünf Schwellwerte berechnet, die als fünf Dreiecke (Polygone) live im vertikalen Bargraph (von unten nach oben) mitwandern. Der Istwert vom Analogeingang `AnalogInput_I4` wird als Zahl angezeigt und steuert über die Hysterese die Ausgänge `Output_Q1` (hoch) und `Output_Q2` (runter). Beim vertikalen Bargraph (Typ „BottomToTop“) wird die Pixelposition gespiegelt: ein hoher Wert ergibt einen kleineren `Top`-Wert; dafür bekommt `Q_ChildPosition_AR_Bargraph_BottomToTop` zusätzlich die Bargraph-Länge (`LengthPx`).

Die Schnittstelle der SubApp ist leer. Die Pool-Objekte stammen aus `Uebungen::const::UT::HystereseVertikal` (`DefaultPool`, `DefaultPool_Numeric`, `DefaultPool_Bargraph`).

Verwendete Funktionsbausteine (FBs)

Eingabefelder und Istwert

Instanz	Typ	Parameter	Erklärung
`MID_N`	NumericValue_PHYSA	`stObj = InputNumber_MID_N`	Eingabefeld Mittelwert (MID), liefert `rPhys`
`DEAD_N`	NumericValue_PHYSA	`stObj = InputNumber_DEAD_N`	Eingabefeld Totzone (DEAD)
`HYST_N`	NumericValue_PHYSA	`stObj = InputNumber_HYSTERESIS_N`	Eingabefeld Hysterese
`AnalogInput_I4`	logiBUS_AI_IDA	`Input = AnalogInput_I4, AnalogInput_hysteresis = DWORD#0, TimeDelta = DWORD#250, TimeRateLimit = DWORD#100`	Analogeingang I4

AD_TO_AUDI	AD_TO_AUDI	–	Adapter-Konvertierung AD ⟶ AUDI
AUDI_TO_AR	AUDI_TO_AR	–	Adapter-Konvertierung AUDI ⟶ AR (REAL)
Mul_Istwert	AR_R_MUL	FACTOR = REAL#0.024420024420024420	Skaliert den Rohwert (Faktor = 100/4095)
Split_Istwert	AR_SPLIT_2	–	Verteilt den Istwert auf Anzeige und Hysterese
Q_Istwert	Q_NumericValue_DHYS	stObj := (u16ObjId := NumberVariable_Istwert, r32Scale := 1.0, i32Offset := 0, u8Decimals := 0)	Zeigt den Istwert im Terminal an

Schwellwerte und Hysterese

Instanz	Typ	Parameter	Erklärung
Thresholds	DualHysteresis_Thresholds_AR	–	Berechnet aus MID, DEAD und HYST die fünf Schwellen (MID, MID±DEAD, MID±(DEAD+HYST))
Split_MID_OUT	AR_SPLIT_2	–	Verteilt MID_OUT auf den Mittelwert-Marker und die Hysterese
DualHysteresis	DualHysteresis_AR_AX	QI = TRUE	Zwei-Wege-Hysterese mit INPUT, MI, DEAD, HYSTERESIS; Ausgänge DO_UP, DO_DOWN
Output_Hoch	logiBUS_QXA	QI = TRUE, Output = Output_Q1	Ausgang „hoch“
Output_Runter	logiBUS_QXA	QI = TRUE, Output = Output_Q2	Ausgang „runter“

Schwellwert-Dreiecke im Bargraph

Instanz	Typ	Parameter (u16ObjId)	Erklärung
QPos_MID	Q_ChildPosition_AR_Bargraph_BottomToTop	Polygon_MID	Marker für MID
QPos_MID_PLUS_DEAD	Q_ChildPosition_AR_Bargraph_BottomToTop	Polygon_MID_PLUS_DEAD	Marker für MID + DEAD
QPos_MID_MINUS_DEAD	Q_ChildPosition_AR_Bargraph_BottomToTop	Polygon_MID_MINUS_DEAD	Marker für MID − DEAD
QPos_MID_PLUS_DEAD_HYST	Q_ChildPosition_AR_Bargraph_BottomToTop	Polygon_MID_PLUS_DEAD_HYST	Marker für MID + (DEAD + HYST)
QPos_MID_MINUS_DEAD_HYST	Q_ChildPosition_AR_Bargraph_BottomToTop	Polygon_MID_MINUS_DEAD_HYST	Marker für MID − (DEAD + HYST)

Gemeinsame Parameter aller fünf QPos_*-Instanzen: u16ObjId, u16ObjIdParent = Container_Schwellwerte, xScale = TRUE, ScalePx = Bargraph_Hysterese_ScalePx, LengthPx = Bargraph_Hysterese_LengthPx.

Programmablauf und Verbindungen

1. **Vorgabe**: `MID_N.rPhys`, `DEAD_N.rPhys` und `HYST_N.rPhys` gehen auf `Thresholds.MI`, `.DEAD` und `.HYST`.
2. **Schwellen → Anzeige**: Die vier abgeleiteten Schwellen (`MID_PLUS_DEAD`, `MID_MINUS_DEAD`, `MID_PLUS_DEAD_HYST`, `MID_MINUS_DEAD_HYST`) gehen jeweils auf `rPhys` des zugehörigen `QPos_*`; `MID_OUT` läuft über `Split_MID_OUT` (`OUT1` → `QPos_MID.rPhys`, `OUT2` → `DualHysteresis.MI`). Verschiebt der Bediener ein Eingabefeld, wandern die Dreiecke live mit.
3. **Hysterese-Parameter**: `Thresholds.DEAD_OUT` → `DualHysteresis.DEAD`, `Thresholds.HYST_OUT` → `DualHysteresis.HYSTERESIS`.
4. **Istwert**: `AnalogInput_I4.IN` → `AD_TO_AUDI` → `AUDI_TO_AR` → `Mul_Istwert` → `Split_Istwert`; `OUT1` → `Q_Istwert.rPhys` (Anzeige), `OUT2` → `DualHysteresis.INPUT`.
5. **Ausgänge**: `DualHysteresis.DO_UP` → `Output_Hoch` (Q1), `DualHysteresis.DO_DOWN` → `Output_Runter` (Q2).

Versionshistorie laut Quelle: V1.1 – AR_TO_AI, Zero und QPos in die SubApp `Q_ChildPosition_AR_*` ausgelagert; V1.2 – Schwellwertberechnung in `DualHysteresis_Thresholds_AR` ausgelagert.

Zusammenfassung

`Uebung_HystereseVertikal_AX_withAnalog` verbindet drei Eingabefelder, eine Analogmessung, eine Zwei-Wege-Hysterese und die Live-Positionierung von fünf Schwellwert-Dreiecken in einem vertikaler Bargraph – komplett über Adapter-Verbindungen, ohne Ereignisverdrahtung.

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

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

2.1.2 Numerischer Index (test_AX)

Hier finden Sie alle Übungen des Arbeitsbereichs `test_AX` in aufsteigender Reihenfolge.

- Bibliothek_Sequenzer_AX
- Uebung_000_AX
- Uebung_000b_AX
- Uebung_001_AX
- Uebung_001_AX_b
- Uebung_001c2_AX
- Uebung_001c3_AX
- Uebung_001c4_AX
- Uebung_001c_AX
- Uebung_001d2_AX
- Uebung_001d_AX
- Uebung_001e_AX
- Uebung_001f_AX
- Uebung_001g_AX
- Uebung_002_AX
- Uebung_002a2_AX
- Uebung_002a3_AX
- Uebung_002a4_AX
- Uebung_002a4b_AX
- Uebung_002a5_AX
- Uebung_002a5b_AX
- Uebung_002a6_AX
- Uebung_002a7_AX
- Uebung_002a_AX
- Uebung_002b2_AX
- Uebung_002b3_AX
- Uebung_003_AX
- Uebung_003a0_AX
- Uebung_003a2b_AX
- Uebung_003a_AX
- Uebung_003a_AX_sub

- Uebung_003b2_AX
- Uebung_003b2_sub_AX
- Uebung_003b3_AX
- Uebung_003b3_sub_AX
- Uebung_003b_AX
- Uebung_003c_AX
- Uebung_003c_sub_AX
- Uebung_003d_AX
- Uebung_003e_AX
- Uebung_003f_AX
- Uebung_004a2_2_AX
- Uebung_004a2_3_AX
- Uebung_004a2_3b_AX
- Uebung_004a2_AX
- Uebung_004a3_AX
- Uebung_004a4_AX
- Uebung_004a4_AX_T
- Uebung_004a5_AX
- Uebung_004a6_AX
- Uebung_004a6a_AX
- Uebung_004a7_AX
- Uebung_004a8_AX
- Uebung_004a9_AX
- Uebung_004a10a_AX
- Uebung_004a10b_AX
- Uebung_004a11a_AX
- Uebung_004a11b_AX
- Uebung_004a_AX
- Uebung_004b2_AX
- Uebung_004b2b_AX
- Uebung_004b3_AX
- Uebung_004b3b_AX
- Uebung_004b3c_AX
- Uebung_004b4c_AX
- Uebung_004b4c_sub_AE
- Uebung_004b4d_AX
- Uebung_004b5_AX
- Uebung_004b6_AX
- Uebung_004b_AX
- Uebung_004b_AX_ASR
- Uebung_004b_AX_ASR_X
- Uebung_004c1_AX
- Uebung_004c2_AX
- Uebung_004c3_AX
- Uebung_004c4_AX
- Uebung_004c5_AX
- Uebung_004c6_AX
- Uebung_004c7_AX
- Uebung_004d_T_AX
- Uebung_005_AX
- Uebung_006_AX
- Uebung_006_SAFE_AX
- Uebung_006_SAFE_AX_02
- Uebung_006a2_AX
- Uebung_006a2b_AX
- Uebung_006a3_AX
- Uebung_006a3_sub_AX
- Uebung_006a4_AX
- Uebung_006a4_AX_RS

- Uebung_006a4_AX_SR
- Uebung_006a5_AX
- Uebung_006a_AX
- Uebung_006b_AX
- Uebung_006c_AX
- Uebung_006d_AX
- Uebung_006e1_AX
- Uebung_006e2_AX
- Uebung_006f_RS_AX
- Uebung_006f_SR_AX
- Uebung_006g_AX
- Uebung_007_AX
- Uebung_007a1_AX
- Uebung_007a2_AX
- Uebung_007a3_AE
- Uebung_007a3_AX
- Uebung_007b_AX
- Uebung_007c_AX
- Uebung_007d2_AX
- Uebung_007d3_AX
- Uebung_007d4_AX
- Uebung_007d_AX
- Uebung_008_AE
- Uebung_008_AX
- Uebung_009_AUDI
- Uebung_009_AX
- Uebung_009a_AX
- Uebung_010_AX
- Uebung_010a2_AX
- Uebung_010a3_AX
- Uebung_010a4_AX
- Uebung_010a5_AX
- Uebung_010a_AX
- Uebung_010b1_AX
- Uebung_010b2_AX
- Uebung_010b3_AX
- Uebung_010b4_AX
- Uebung_010b4_sub_AX
- Uebung_010b5_AX
- Uebung_010b5_sub_AX
- Uebung_010b6_AX
- Uebung_010b7_AX
- Uebung_010b8_AX
- Uebung_010b9_AX
- Uebung_010bA2_AX
- Uebung_010bA3_AX
- Uebung_010bA4_AX
- Uebung_010bA_AX
- Uebung_010c2_AX
- Uebung_010c3_AX
- Uebung_010c3_sub_AX
- Uebung_010c4_AX
- Uebung_010c4_sub_AX
- Uebung_010c_AX
- Uebung_010d2_AX
- Uebung_010d2_sub_AX
- Uebung_010d_AX
- Uebung_010e2_AX
- Uebung_010e2_sub_AX

- Uebung_010e_AX
- Uebung_010f2_AX
- Uebung_010f2_sub_AX
- Uebung_010f3_AX
- Uebung_010f4_AX
- Uebung_010f_AX
- Uebung_011_AUDI
- Uebung_011_AX
- Uebung_011a2_AUDI
- Uebung_011a2_AX
- Uebung_011a_AUDI
- Uebung_011a_AX
- Uebung_011b1_AUDI
- Uebung_011b1_AX
- Uebung_011b1_PHYS_AX
- Uebung_011b1_PHYSA
- Uebung_011b2_AX
- Uebung_011b3_AX
- Uebung_011b3_PHYS_AX
- Uebung_011b3_PHYSA
- Uebung_011b4_AX
- Uebung_011b5_AX
- Uebung_011b6_AX
- Uebung_011b7_AX
- Uebung_011c_AUDI
- Uebung_011c_AX
- Uebung_011d_AUDI
- Uebung_011d_AX
- Uebung_011e_MIX_AX
- Uebung_011e_MIXA
- Uebung_011f_AX
- Uebung_011f_PHYSA
- Uebung_011f_ROUND_AR_AX
- Uebung_011g1_AX
- Uebung_012_AUDI
- Uebung_012_AX
- Uebung_012a_AX
- Uebung_012b_AUDI
- Uebung_012b_AX
- Uebung_012c_AUDI
- Uebung_012c_AX
- Uebung_012d_AR
- Uebung_012d_AR2
- Uebung_012d_AX
- Uebung_012e2_AR
- Uebung_012e_AR
- Uebung_012e_AR2
- Uebung_012e_AX
- Uebung_012f_AR
- Uebung_012f_AR2
- Uebung_012f_AX
- Uebung_012g_AR
- Uebung_012g_AR2
- Uebung_012g_AX
- Uebung_012h_AR2
- Uebung_012i_AR2
- Uebung_012j_AIS
- Uebung_012j_AX
- Uebung_012k_AIS

- Uebung_012k_AX
- Uebung_012l_AIS
- Uebung_012l_AX
- Uebung_012m_AIS
- Uebung_012m_AX
- Uebung_012n_AR2
- Uebung_012o_AR2
- Uebung_013_AX
- Uebung_014_AX
- Uebung_014_AXa
- Uebung_014_AXb
- Uebung_015_AX
- Uebung_015a_AX
- Uebung_015b_AX
- Uebung_016_AX
- Uebung_016a_AX
- Uebung_016b_AX
- Uebung_016c_AX
- Uebung_017_AX
- Uebung_018_AX
- Uebung_018a_AX
- Uebung_019_AX
- Uebung_019a_AX
- Uebung_019b_AX
- Uebung_019c_AX
- Uebung_020a_AX
- Uebung_020b_AX
- Uebung_020c2_AX
- Uebung_020c3_AX
- Uebung_020c4_AX
- Uebung_020c_AX
- Uebung_020d_AX
- Uebung_020e2_AX
- Uebung_020e_AX
- Uebung_020f2_AX
- Uebung_020f3_AX
- Uebung_020f4_AX
- Uebung_020f_AX
- Uebung_020g_AX
- Uebung_020h_AX
- Uebung_020i_AX
- Uebung_020j2_AX
- Uebung_020j2_AX_sub
- Uebung_020j_AX
- Uebung_021_AX
- Uebung_021b_AX2
- Uebung_022_AX
- Uebung_022b_AX2
- Uebung_023_AX
- Uebung_023b_AX2
- Uebung_024_AX
- Uebung_025_AX
- Uebung_026_AX
- Uebung_028_AX
- Uebung_028a2_AR
- Uebung_028a_AR
- Uebung_028a_AX
- Uebung_028b2_AR
- Uebung_028b_AR

- Uebung_028c2_AR
- Uebung_028c3_AR
- Uebung_028c_AR
- Uebung_029_AX
- Uebung_030_AX
- Uebung_031_AX
- Uebung_032_AX
- Uebung_033_AX
- Uebung_034_AD
- Uebung_034_AX
- Uebung_034a1_Q1_AX
- Uebung_034a1_Q2_AX
- Uebung_034a1_Q4_AX
- Uebung_034b_AUDI
- Uebung_034b_AX
- Uebung_035_AX
- Uebung_035a1_AX
- Uebung_035a1b_AX
- Uebung_035a2_AX
- Uebung_035a3_AX
- Uebung_035a_AX
- Uebung_035a_AX_Beschreibung
- Uebung_035b_AX
- Uebung_035c_AX
- Uebung_036_AX
- Uebung_036_AX_Beschreibung
- Uebung_036b_AX
- Uebung_036b_AX_Beschreibung
- Uebung_037_AX
- Uebung_037_AX_Beschreibung
- Uebung_038_AX
- Uebung_039_AX
- Uebung_039_sub_NumbAnzeig_AS
- Uebung_039_sub_NumbAnzeig_AX
- Uebung_039_sub_Outputs_AX
- Uebung_039a_AX
- Uebung_039b_AX
- Uebung_040_2_AX
- Uebung_040_AX
- Uebung_041_AX
- Uebung_042_AX
- Uebung_043_AX
- Uebung_044_AX
- Uebung_044_AX_Beschreibung
- Uebung_045_AX
- Uebung_045_AX_Beschreibung
- Uebung_046_AX
- Uebung_046_AX_Beschreibung
- Uebung_047_AX
- Uebung_047_AX_Beschreibung
- Uebung_048_AX
- Uebung_048_AX_Beschreibung
- Uebung_049_AX
- Uebung_051_AX
- Uebung_052_AX
- Uebung_053_AB
- Uebung_053_AX
- Uebung_054_AX
- Uebung_055_AX

- Uebung_056_AX
- Uebung_057_AX
- Uebung_057_AX_Beschreibung
- Uebung_058_AX
- Uebung_058_AX_Beschreibung
- Uebung_059_AX
- Uebung_059_AX_Beschreibung
- Uebung_060_AX
- Uebung_061_AX
- Uebung_061_AX_Beschreibung
- Uebung_062_AX
- Uebung_062_AX_Beschreibung
- Uebung_063_AX
- Uebung_063_AX_Beschreibung
- Uebung_064_AX
- Uebung_064_AX_Beschreibung
- Uebung_070_AUI
- Uebung_070_AX
- Uebung_070c_AUI
- Uebung_070d_AUI
- Uebung_071_AUI
- Uebung_071a_AUI
- Uebung_071b_AUI
- Uebung_072_AUI
- Uebung_072c_AUI
- Uebung_072d_AUI
- Uebung_073_AUI
- Uebung_073c_AUI
- Uebung_073d_AUI
- Uebung_074_AUI
- Uebung_074a_AUI
- Uebung_074b_AUI
- Uebung_074c_AUI
- Uebung_075_AX
- Uebung_075a_AUI
- Uebung_076_AUI
- Uebung_076a_AUI
- Uebung_076b_AUI
- Uebung_076c_AUI
- Uebung_077_AUI
- Uebung_078_AUI
- Uebung_079_AX
- Uebung_079a_AUI
- Uebung_080_AUI
- Uebung_080_AX
- Uebung_080b_AUI
- Uebung_080b_AX
- Uebung_080c_AUI
- Uebung_080c_AX
- Uebung_080d_AX
- Uebung_080e2_AX
- Uebung_080e3_AX
- Uebung_080e4_AX
- Uebung_080e_AX
- Uebung_080f_AX
- Uebung_081_AX
- Uebung_082_AX
- Uebung_083_AUDI
- Uebung_083_AX

- Uebung_084_AX
- Uebung_085_AX
- Uebung_086_AX
- Uebung_087_AX
- Uebung_087a1_AX
- Uebung_087a2_AX
- Uebung_088_AX
- Uebung_089_AX
- Uebung_089a_AX
- Uebung_090a1_AX
- Uebung_090a2_AX
- Uebung_091_AX
- Uebung_093_AX
- Uebung_093b_AX
- Uebung_094_AX
- Uebung_094a_AX
- Uebung_094b2_AX
- Uebung_094b3_AX
- Uebung_094b3b_AX
- Uebung_094b3c_AX
- Uebung_094b3d_AX
- Uebung_094b3e_AX
- Uebung_094b3f_AX
- Uebung_094b4_AX
- Uebung_094b4b_AX
- Uebung_094b5_AX
- Uebung_094b5b_AX
- Uebung_094b_AX
- Uebung_094c2_AX
- Uebung_094c2b_AX
- Uebung_094c_AX
- Uebung_094d_AX
- Uebung_095_AX
- Uebung_103
- Uebung_103c
- Uebung_103c2
- Uebung_104_AX
- Uebung_110_AX
- Uebung_111_AX
- Uebung_120_AX
- Uebung_121_AX
- Uebung_122_AX
- Uebung_122b_AX
- Uebung_123_AX
- Uebung_124_AX
- Uebung_125_AX
- Uebung_126_AX
- Uebung_126b2_AX
- Uebung_126b_AX
- Uebung_127_AX
- Uebung_128_AX
- Uebung_128b_AX
- Uebung_130_AX
- Uebung_131_AX
- Uebung_132_AX
- Uebung_133_AX
- Uebung_134_AX
- Uebung_135_AX
- Uebung_140_AX

- Uebung_150_AX
- Uebung_151_AX
- Uebung_152_AX
- Uebung_152_AX_Beschreibung
- Uebung_153_AX
- Uebung_153_AX_Beschreibung
- Uebung_160_AX
- Uebung_160b2_AX
- Uebung_160b_AX
- Uebung_170_AX
- Uebung_171_ASR
- Uebung_171_AX
- Uebung_171b_ASR
- Uebung_171c_ASR
- Uebung_172_ASRT
- Uebung_172_AX
- Uebung_172b_ASRT
- Uebung_172c_ASRT
- Uebung_173_AX
- Uebung_174_AX
- Uebung_175_AX
- Uebung_177_AX
- Uebung_178_AX
- Uebung_179_AX
- Uebung_180_AX
- Uebung_201_AX
- Uebung_201_Interlock_AX
- Uebung_201_Interlock_BOOL_AX
- Uebung_201b_AX
- Uebung_202_AX
- Uebung_202b_AX
- Uebung_203_AX
- Uebung_203b_AX
- Uebung_204_AX
- Uebung_204b_AX
- Uebung_204c_AX
- Uebung_205_AX
- Uebung_205b_AX
- Uebung_206_AX
- Uebung_206b_AX
- Uebung_207_AX
- Uebung_208_AX
- Uebung_209_AX
- Uebung_210_AI
- Uebung_210b_AR
- Uebung_211_ADI
- Uebung_211b_ALR
- Uebung_212_ALI
- Uebung_212b_ALR
- Uebung_213_AUDI
- Uebung_213b_ALR
- Uebung_214_AULI
- Uebung_214b_ALR
- Uebung_215_AI
- Uebung_215b_AR
- Uebung_216_ADI
- Uebung_216b_ALR
- Uebung_217_ALI
- Uebung_217b_ALR

- Uebung_218_AUDI
- Uebung_218b_ALR
- Uebung_219_AULI
- Uebung_219b_ALR
- Uebung_220_AI
- Uebung_220b_AR
- Uebung_221_ADI
- Uebung_221b_ALR
- Uebung_222_ALI
- Uebung_222b_ALR
- Uebung_223_AUDI
- Uebung_223b_ALR
- Uebung_224_AULI
- Uebung_224b_ALR
- Uebung_225_AX
- Uebung_225_AX_Beschreibung
- Uebung_225b_AX
- Uebung_225b_AX_Beschreibung
- Uebung_226_AX
- Uebung_226_AX_Beschreibung
- Uebung_227_AX
- Uebung_227_AX_Beschreibung
- Uebung_228_AX
- Uebung_228_AX_Beschreibung
- Uebung_229_AX
- Uebung_229_AX_Beschreibung
- Uebung_230_AX
- Uebung_230_AX_Beschreibung
- Uebung_231_AX
- Uebung_232_AX
- Uebung_233_AX
- Uebung_234_AX
- Uebung_235_AX
- Uebung_235_AX_Beschreibung
- Uebung_236_AX
- Uebung_236_AX_Beschreibung
- Uebung_237_AX
- Uebung_245_AX
- Uebung_245_AX_Beschreibung
- Uebung_246_AX
- Uebung_246_AX_Beschreibung
- Uebung_247_AX
- Uebung_247_AX_Beschreibung
- Uebung_248_AX
- Uebung_248_AX_Beschreibung
- Uebung_249_AX
- Uebung_249b_AX
- Uebung_Isobus_Softkeys_Index
- Uebungen_Basis_Index
- Uebungen_doc_Index
- Uebungen_FlipFlops_Index
- Uebungen_Index
- Uebungen_Sequenzer_Index
- Uebungen_Signale_Index
- Uebungen_SubApps_Index
- Uebungen_Timer_Index

2.1.2.1 📚 Bibliotheken: Sequenzer (AX Adapter)

Dieser Abschnitt beschreibt die in der logiBUS®-Bibliothek verfügbaren Sequenzer-Bausteine, die für die Verwendung mit dem **AX Adapter** optimiert sind.

---

i Allgemeines

Die Sequenzer-Bausteine ermöglichen die Implementierung von Schrittketten (Sequential Function Charts). Die AX-Versionen zeichnen sich dadurch aus, dass sie **AX-Adapter** (Plug/Socket) für die Ansteuerung der Ausgänge nutzen. Dies reduziert den Verdrahtungsaufwand erheblich, da Status-Events und Datenbits in einer Verbindung gebündelt werden.

⚙️ Event-gesteuerte Sequenzer (`sequence_ET_xx_AX`)

Diese Bausteine schalten zum nächsten Schritt, wenn entweder ein externes Ereignis (`S1_S2`, `S2_S3`, etc.) eintritt oder eine definierte Zeit (`DT_S*_S*`) abgelaufen ist.

Verfügbare Typen (Event-gesteuert)

- `sequence_ET_04_AX`: 4 Schritte, 4 Ausgänge.
- `sequence_ET_04_loop_AX`: 4 Schritte, zyklisch (loop).
- `sequence_ET_05_AX`: 5 Schritte, 5 Ausgänge.
- `sequence_ET_05_loop_AX`: 5 Schritte, zyklisch.
- `sequence_ET_08_AX`: 8 Schritte, 8 Ausgänge.
- `sequence_ET_08_loop_AX`: 8 Schritte, zyklisch.

⚙️ Zeit-gesteuerte Sequenzer (`sequence_T_xx_AX`)

Diese Bausteine schalten automatisch nach Ablauf der für den jeweiligen Schritt definierten Zeit weiter.

Verfügbare Typen (Zeit-gesteuert)

- `sequence_T_04_AX`: 4 Schritte, zeitgesteuert.
- `sequence_T_04_loop_AX`: 4 Schritte, zyklisch.
- `sequence_T_05_AX`: 5 Schritte, zeitgesteuert.
- `sequence_T_05_loop_AX`: 5 Schritte, zyklisch.
- `sequence_T_08_AX`: 8 Schritte, zeitgesteuert.
- `sequence_T_08_loop_AX`: 8 Schritte, zyklisch.

⚙️ Pattern-Sequenzer (Nockenschaltwerke)

Die Pattern-Sequenzer (`sequence_Pattern_xx_AX`) ermöglichen es, für jeden Schritt ein individuelles Bitmuster (Pattern) für die Ausgänge zu definieren. Dies ist ideal für Anwendungen, bei denen in einem Schritt mehrere Aktoren gleichzeitig in bestimmten Mustern geschaltet werden müssen.

Verfügbare Typen (Pattern)

- **`sequence_Pattern_04_04_loop_AX`**: 4 Schritte, 4 Ausgänge, zyklisch.
- **`sequence_Pattern_08_08_loop_AX`**: 8 Schritte, 8 Ausgänge, zyklisch.

Funktionsweise

Jeder Zustand `S1` bis `Sx` hat einen zugehörigen Eingang `P_S1` bis `P_Sx` vom Typ `BYTE`. Die einzelnen Bits dieses Bytes steuern die Adapter-Ausgänge `Q1` bis `Qx`:

- Bit 0 -> Ausgang Q1
- Bit 1 -> Ausgang Q2
- ...und so weiter.

Beim Übergang in einen neuen Zustand wird das entsprechende Muster sofort an die Ausgänge angelegt.

⚙️ Reine Event-Sequenzer (`sequence_E_xx_AX`)

Im Gegensatz zur `ET`-Serie besitzen diese Bausteine **keine integrierte Zeitüberwachung**. Ein Weiterschalten erfolgt ausschließlich durch externe Ereignis-Signale.

Verfügbare Typen (Event, ohne Zeitüberwachung)

- `sequence_E_04_AX`: 4 Schritte.
- `sequence_E_04_loop_AX`: 4 Schritte, zyklisch.
- `sequence_E_05_AX`: 5 Schritte.
- `sequence_E_05_loop_AX`: 5 Schritte, zyklisch.
- `sequence_E_08_AX`: 8 Schritte.
- `sequence_E_08_loop_AX`: 8 Schritte, zyklisch.

⚙️ Spezialisierte Sequenz-Bausteine

Neben den universellen Schrittketten gibt es spezialisierte Bausteine für häufig vorkommende Aufgabenstellungen.

Schiebersteuerung (`SchieberControl_AX`)

Dieser Baustein dient zur sequentiellen Ansteuerung von Schiebern oder Ventilen mit Endlagenrückmeldung. Er bündelt die Logik für das Öffnen/Schließen und die Überwachung der Laufzeit.

Links-Rechts-Verteiler (`LinksRechts_AX`)

Spezialbaustein für Anwendungen wie z.B. Förderbänder oder Weichen, die zwischen zwei Richtungen oder Zuständen hin- und herschalten müssen, inklusive der notwendigen Verriegelungen und Umschaltpausen.

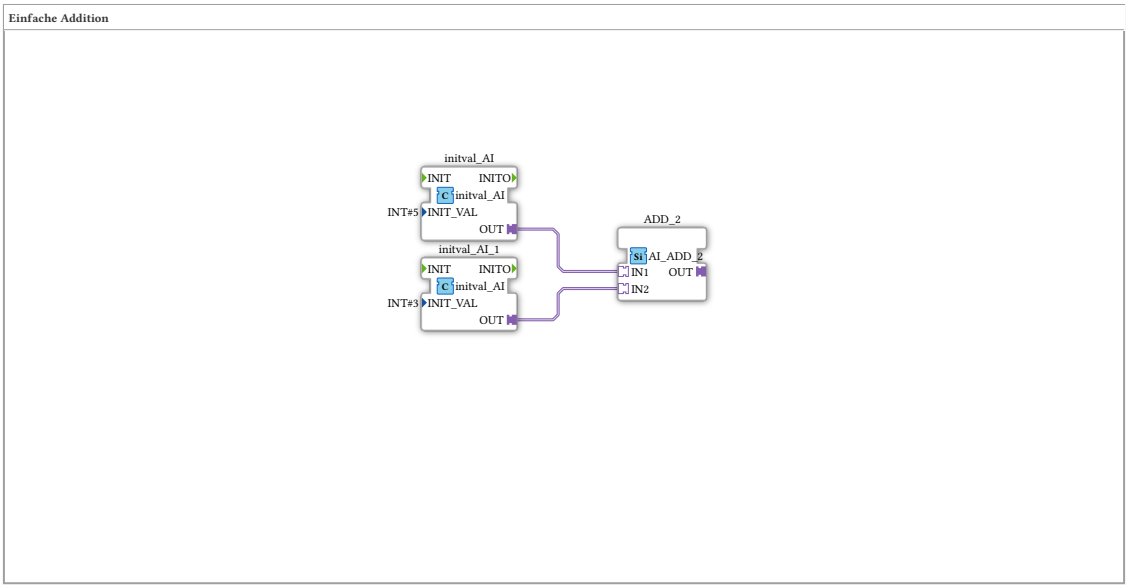
🔌 Schnittstellenbeschreibung (Beispiel AX)

Anschluss	Typ	Beschreibung
Plugs (Ausgänge)	`adapter::types::unidirectional::AX`	Gebündelte Ausgänge für Daten und Events.
START_S1	`Event`	Startet die Sequenz im ersten Schritt.
RESET	`Event`	Setzt die Sequenz sofort auf den Initialzustand (START) zurück.
STATE_NR	`SINT`	Aktuelle Schrittnummer (0 = START, 1 = S1, ...).
timeOut	`adapter`	Adapter zur Anbindung eines Time-Out-Managements.

🛠️ Zugehörige Übungen

- Uebung_035a1_AX (Nutzung von `sequence_Pattern_04_04_loop_AX`)
- Uebung_039_AX (Nutzung von `sequence_ET_05_AX`)

2.1.2.2 Uebung_000_AX: Einfache Addition



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung ist das absolute Basisbeispiel für Berechnungen: Zwei konstante Ganzzahlwerte werden über Adapter-Verbindungen an einen Additionsbaustein der adapterbasierten IEC-61131-Bibliothek geführt. Sie zeigt, wie ein Standard-Rechenbaustein innerhalb eines IEC 61499 Netzwerks platziert und mit Werten versorgt wird. Die Übung hat keine Ein- und Ausgänge an der SubApp-Schnittstelle.

Verwendete Funktionsbausteine (FBs)

- **initval_AI**: Konstantwert-Quelle für den ersten Summanden
  - **Typ**: `adapter::types::unidirectional::AI::initval::initval_AI`
  - **Parameter**: `INIT_VAL = INT#5`
  - **Erklärung**: Gibt beim Initialisieren den Wert 5 über den Adapter-Ausgang `OUT` (Adapter `AI`) aus.
- **initval_AI_1**: Konstantwert-Quelle für den zweiten Summanden
  - **Typ**: `adapter::types::unidirectional::AI::initval::initval_AI`
  - **Parameter**: `INIT_VAL = INT#3`
  - **Erklärung**: Gibt beim Initialisieren den Wert 3 über den Adapter-Ausgang `OUT` aus.
- **ADD_2**: Additionsbaustein
  - **Typ**: `adapter::iec61131::arithmetic::AI_ADD_2`
  - **Parameter**: keine
  - **Erklärung**: Addiert die über die Adapter `IN1` und `IN2` (jeweils Typ `AI`) angelieferten Werte und stellt die Summe am Adapter `OUT` bereit. Der Adapter-Ausgang wird laut Typ-Kommentar nur bei einer Wertänderung aktualisiert.

Programmablauf und Verbindungen

1. `initval_AI.OUT` ⟶ `ADD_2.IN1` (AdapterConnection): der Wert 5 liegt am ersten Summanden an.
2. `initval_AI_1.OUT` ⟶ `ADD_2.IN2` (AdapterConnection): der Wert 3 liegt am zweiten Summanden an.

Der Ausgang `ADD_2.OUT` ist nicht weiterverschaltet; das Ergebnis (5 + 3 = 8) liegt dort an und kann zur Laufzeit beobachtet werden.

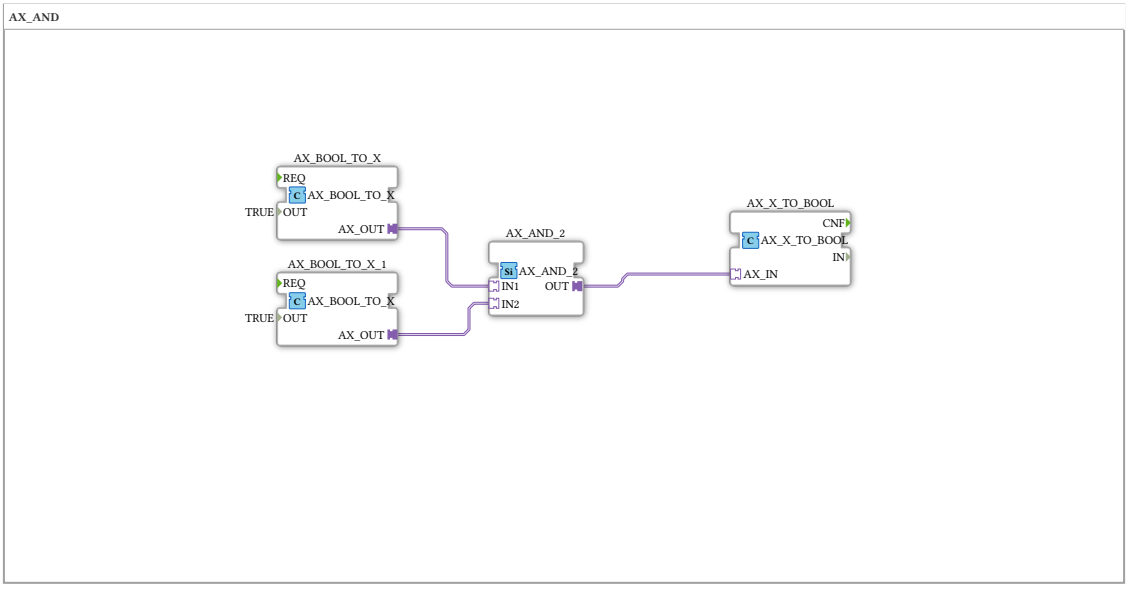
Zusammenfassung

Die Übung `Uebung_000_AX` ist die einfachste Berechnung im Übungsumfeld: Zwei `initval_AI`-Bausteine liefern die Konstanten 5 und 3, `AI_ADD_2` addiert sie über Adapter-Verbindungen zu 8. Sie ist die Grundlage für jede Form von Zählern, Offsets oder Skalierungen in einer Steuerung.

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

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

2.1.2.3 Uebung_000b_AX: AX_AND



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung demonstriert die Verwendung von **Adapter-Funktionsbausteinen** zur Realisierung einer logischen UND-Verknüpfung. Innerhalb der SubApp werden Boolesche Werte über Adapterverbindungen zwischen Konvertierungsbausteinen und einem logischen UND-Baustein ausgetauscht. Sie lernen die grundlegende Struktur von Adaptern und deren Zusammenspiel in einer SubApp kennen.

Die Übung bildet eine einfache UND-Logik ab, bei der zwei konstante **TRUE**-Werte über den `AX_AND_2`-Baustein verknüpft werden. Das Ergebnis wird über einen weiteren Konverter ausgegeben.

Verwendete Funktionsbausteine (FBs)

Die SubApp besteht aus drei internen Funktionsbausteinen:

AX_BOOL_TO_X (zweimal instanziiert)

- **Typ**: `adapter::conversion::unidirectional::AX_BOOL_TO_X`
- **Parameter**:
  - `OUT` = `TRUE` (setzt den Ausgang auf den Wert **TRUE**)
- **Funktion**: Konvertiert einen Booleschen Wert (über Parameter oder Eingang) in den Adapter-Datentyp `X`. In dieser Übung wird der Ausgang direkt auf `TRUE` gesetzt, sodass der Baustein als Konstantengeber fungiert.
- **Verbindungen**:
  - `AX_BOOL_TO_X.AX_OUT` ⟶ `AX_AND_2.IN1`
  - `AX_BOOL_TO_X_1.AX_OUT` ⟶ `AX_AND_2.IN2`

AX_AND_2

- **Typ**: `adapter::booleanOperators::AX_AND_2`
- **Parameter**: Keine
- **Funktion**: Führt eine logische UND-Verknüpfung der beiden Eingänge (`IN1`, `IN2`) durch. Der Ausgang (`OUT`) wird nach Ausführung der UND-Operation gesetzt.
- **Verbindungen**:
  - Eingänge: von `AX_BOOL_TO_X` (IN1) und `AX_BOOL_TO_X_1` (IN2)
  - Ausgang (`OUT`) ⟶ `AX_X_TO_BOOL.AX_IN`

AX_X_TO_BOOL

- **Typ**: `adapter::conversion::unidirectional::AX_X_TO_BOOL`
- **Parameter**: Keine
- **Funktion**: Konvertiert einen Wert des Adapter-Datentyps `X` zurück in einen Booleschen Wert. Dies stellt den Ausgang der Gesamtschaltung dar.
- **Verbindungen**:
  - Eingang (`AX_IN`) von `AX_AND_2.OUT`

Programmablauf und Verbindungen

Die SubApp besitzt keine eigenen externen Schnittstellen (keine Ein-/Ausgänge auf SubApp-Ebene). Die gesamte Logik ist intern fest verdrahtet:

1. Zwei `AX_BOOL_TO_X`-Bausteine liefern konstant den Wert **TRUE**.
2. Diese Werte werden über Adapter-Verbindungen an die beiden Eingänge des `AX_AND_2`-Bausteins übergeben.
3. Der `AX_AND_2`-Baustein berechnet die UND-Verknüpfung: `TRUE AND TRUE = TRUE`.
4. Das Ergebnis wird über eine Adapter-Verbindung zum `AX_X_TO_BOOL`-Baustein weitergeleitet, der den Wert zurück in einen Booleschen Wert wandelt.

Da alle Eingangswerte `TRUE` sind, liegt am Ausgang des `AX_X_TO_BOOL`-Bausteins stets **TRUE** an. In einer erweiterten Anwendung könnten die konstanten Werte durch externe Signale ersetzt werden, indem die SubApp mit entsprechenden Adapter-Schnittstellen ergänzt wird.

Lernziele:

- Verständnis des Zusammenspiels von Adaptern in 4diac.
- Unterschied zwischen Datenflüssen und Adapterverbindungen.
- Grundlagen der Booleschen Logik mit `AX_AND_2`.

Schwierigkeitsgrad: Einfach

Benötigte Vorkenntnisse: Grundlegende Bedienung der 4diac IDE, Kenntnis der Baustein-Typen „Adapter“ und „Funktionsbaustein“.

Ausführung: Die Übung kann ohne externe Signale in einer Simulation getestet werden – der Ausgang bleibt konstant TRUE. Um dynamische Werte zu verarbeiten, müssten die konstanten `AX_BOOL_TO_X`-Bausteine durch Adapter-Eingänge ersetzt werden.

Zusammenfassung

Die Übung `Uebung_000b_AX` zeigt eine einfache UND-Logik, die ausschließlich mit Adapter-Funktionsbausteinen realisiert wird. Sie verdeutlicht die Notwendigkeit von Datentyp-Konvertierungen (`AX_BOOL_TO_X` und `AX_X_TO_BOOL`) und die direkte Verbindung von Adapterausgängen mit Adaptereingängen. Durch die Verwendung von konstanten Werten wird das prinzipielle Verhalten demonstriert, das später durch dynamische Signale erweitert werden kann.

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

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

2.1.2.4 Uebung_001_AX: DigitalInput_I1 auf DigitalOutput_Q1, mit Plug and Socket

NotebookLM

Dieser Artikel beschreibt die grundlegende logiBUS®-Übung `Uebung_001_AX`, bei der ein digitaler Eingang direkt mit einem digitalen Ausgang verbunden wird, unter Verwendung des Adapters AX.





Ziel der Übung

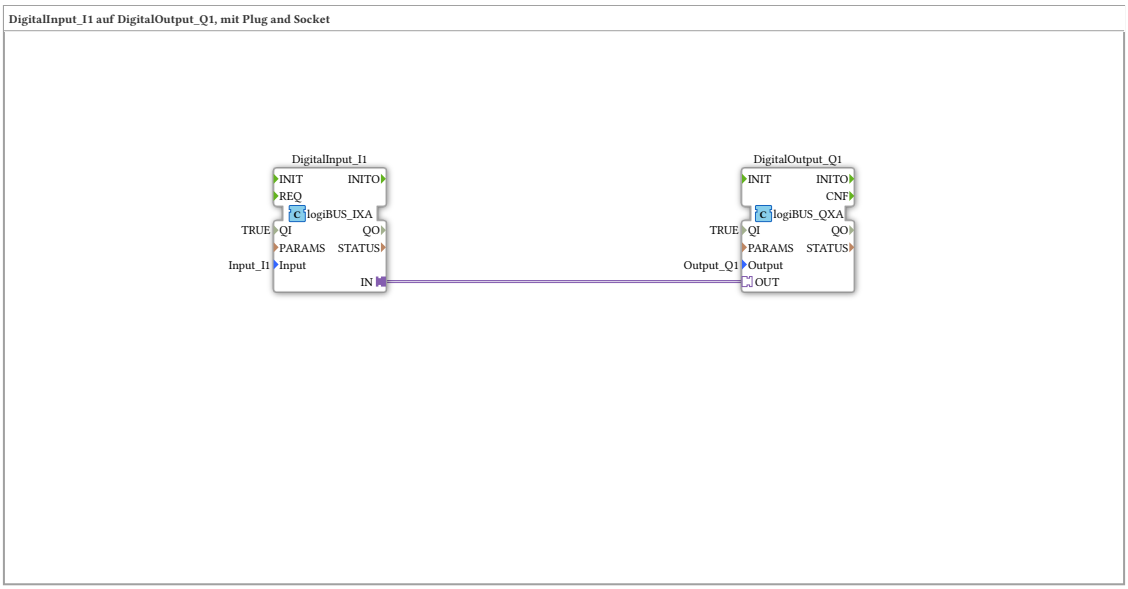
Das Hauptziel dieser Übung ist es, das grundlegende Prinzip der direkten Signalweiterleitung von einem physischen digitalen Eingang zu einem physischen digitalen Ausgang zu demonstrieren. Dabei wird das „Plug and Socket"-Konzept der IEC 61499 mittels einer Adapter-Schnittstelle verwendet. Die Logik ist denkbar einfach: Der Zustand des Ausgangs soll immer dem Zustand des Eingangs entsprechen.

Beschreibung und Komponenten

[cite_start]Die Übung besteht aus einer Subapplikation (`Uebung_001_AX.SUB`), die zwei Funktionsbausteine über eine Adapterverbindung miteinander verknüpft. [cite_start]Als Schnittstelle für diese Verbindung dient der Adapter-Typ `AX.adp`.

Funktionsbausteine (FBs)

In der Subapplikation werden zwei zentrale Funktionsbausteine instanziiert:



- **`DigitalInput_I1`**: Eine Instanz des Typs `logiBUS_IXA`. Dieser Baustein repräsentiert einen physischen digitalen Eingang. [cite_start]Über den Parameter `Input` wird er fest mit dem Hardware-Eingang `logiBUS_DI::Input_I1` verdrahtet.
- **`DigitalOutput_Q1`**: Eine Instanz des Typs `logiBUS_QXA`. Dieser Baustein repräsentiert einen physischen digitalen Ausgang. [cite_start]Sein Parameter `Output` verweist auf den Hardware-Ausgang `logiBUS_DO::Output_Q1`.

Adapter-Schnittstelle: `AX.adp`

Die Verbindung zwischen den beiden Bausteinen wird durch den Adapter-Typ `AX` realisiert. [cite_start]Es handelt sich um eine unidirektionale (einseitig gerichtete) Schnittstelle, die definiert ist, um genau ein Ereignis und einen dazugehörigen booleschen Datenwert zu übertragen.

- [cite_start]**Event `E1`**: Ein Ereignis, das eine Zustandsänderung signalisiert.
- [cite_start]**Variable `D1` (BOOL)**: Der boolesche Wert (wahr/falsch), der mit dem Ereignis `E1` gesendet wird.

Funktionsweise

Die Logik wird ausschließlich durch die Verbindung der beiden Bausteine realisiert. In der Subapplikation `Uebung_001_AX.SUB` wird der „Plug“ des Eingangsbausteins mit der „Socket“ des Ausgangsbausteins verbunden:

```
<AdapterConnections>
    <Connection Source="DigitalInput_I1.IN" Destination="DigitalOutput_Q1.OUT"/>
</AdapterConnections>
```

Dieser einzelnen Zeile Code implementiert die gesamte Funktionalität:

1. Der Baustein `DigitalInput_I1` liest kontinuierlich den Zustand des physischen Eingangs `Input_I1`.
2. Sobald sich der Zustand des Eingangs ändert, sendet der Baustein über seinen `IN`-Adapter-Anschluss (den „Plug“) das Ereignis `E1` zusammen mit dem aktuellen booleschen Zustand (`D1`).
3. Der Baustein `DigitalOutput_Q1` empfängt dieses Ereignis und den Datenwert an seinem `OUT`-Anschluss (der „Socket“).
4. Unmittelbar nach dem Empfang setzt der `DigitalOutput_Q1`-Baustein den physischen Ausgang `Output_Q1` auf den empfangenen Wert von `D1`.

Im Ergebnis spiegelt der **digitale Ausgang Q1** exakt und in Echtzeit den Zustand des **digitalen Eingangs I1**.

Anwendungsbeispiel

Diese Konfiguration ist die einfachste Form einer Steuerungsanwendung und dient oft als „Hallo Welt“-Beispiel für die Hardware-Anbindung in logiBUS®. Ein praktischer Anwendungsfall wäre ein einfacher **Funktionstest** für die Verdrahtung:

- An `Input_I1` wird ein Schalter angeschlossen.
- An `Output_Q1` wird eine Lampe angeschlossen.

Wird der Schalter betätigt, muss die Lampe sofort aufleuchten. Dies bestätigt, dass sowohl der Eingangs- als auch der Ausgangskanal korrekt konfiguriert und verdrahtet sind.

2.1.2.5 Uebung_001_AX_b: DigitalInput_I1 auf DigitalOutput_Q1, mit Plug and Socket



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_001_AX_b`, bei der ein digitaler Eingang über eine Signalwandlung mit einem digitalen Ausgang verbunden wird. Im Gegensatz zur direkten Adapterverbindung wird hier der Adapter-Zustand explizit in einen booleschen Wert und wieder zurück gewandelt.

Ziel der Übung

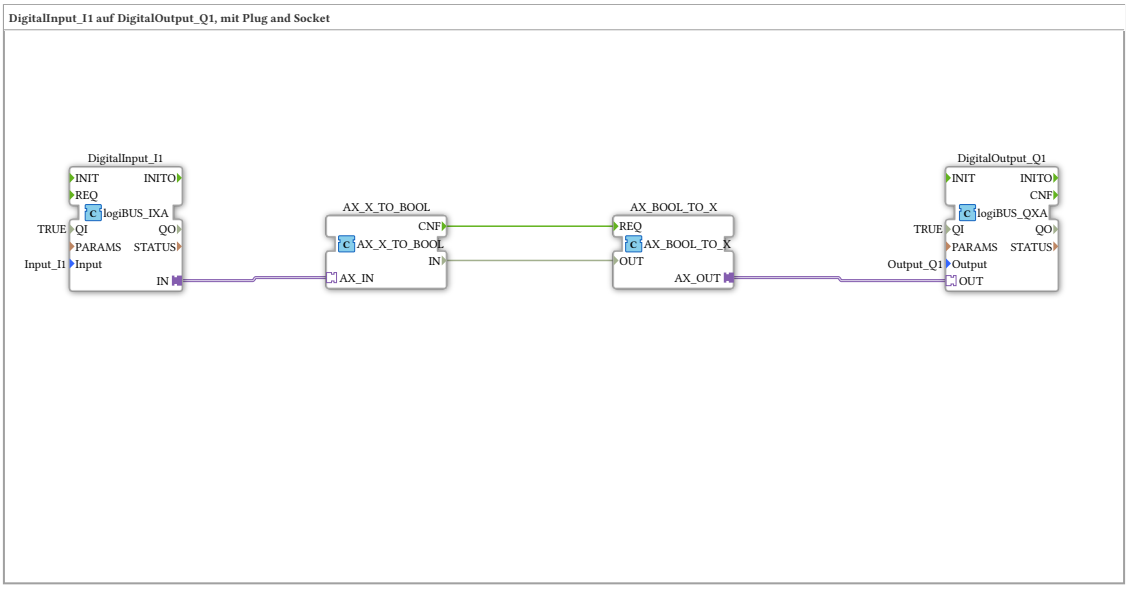
Das Hauptziel dieser Übung ist es, die Wandlung zwischen Adapter-Schnittstellen („Plug and Socket“) und klassischen booleschen Datenverbindungen zu demonstrieren. Dies ist eine grundlegende Technik in der IEC 61499, um Signale, die über Adapter übertragen werden, für logische Operationen (wie AND, OR, NOT) zugänglich zu machen, die auf elementaren Datentypen operieren.

Beschreibung und Komponenten

Die Übung basiert auf der Subapplikation `Uebung_001_AX_b.SUB`, die den Signalfluss von einem digitalen Eingang zu einem digitalen Ausgang über zwei zwischengeschaltete Wandler-Bausteine realisiert.

Funktionsbausteine (FBs)

In der Subapplikation werden vier Funktionsbausteine instanziiert:



- **`DigitalInput_I1`**: Eine Instanz des Typs `logiBUS_IXA`. Dieser Baustein liest den Zustand des physischen Eingangs `Input_I1` und stellt ihn über seinen Adapter-Anschluss `IN` bereit.
- **`DigitalOutput_Q1`**: Eine Instanz des Typs `logiBUS_QXA`. Dieser Baustein empfängt Signale an seinem Adapter-Anschluss `OUT` und setzt entsprechend den physischen Ausgang `Output_Q1`.
- **`AX_X_TO_BOOL`**: Ein Wandler-Baustein, der ein am Adapter-Eingang `AX_IN` (Socket) empfangenes Signal in ein Ereignis `CNF` und einen booleschen Datenwert `IN` zerlegt.

- **AX_BOOL_TO_X**: Ein Wandler-Baustein, der aus einem Ereignis REQ und einem booleschen Datenwert OUT wieder ein Adapter-Signal am Ausgang AX_OUT (Plug) zusammensetzt.

Adapter-Schnittstelle: AX.adp

Wie in der Basisübung dient auch hier der Adapter-Typ AX als Schnittstelle, der das Ereignis E1 und den booleschen Wert D1 überträgt.

Funktionsweise

Die Logik wird durch die Verknüpfung von Adapter-, Ereignis- und Datenverbindungen realisiert. Der Signalweg ist in der Datei Uebung_001_AX_b.SUB wie folgt definiert:

```
<EventConnections>
    <Connection Source="AX_X_TO_BOOL.CNF" Destination="AX_BOOL_TO_X.REQ"/>
</EventConnections>
<DataConnections>
    <Connection Source="AX_X_TO_BOOL.IN" Destination="AX_BOOL_TO_X.OUT"/>
</DataConnections>
<AdapterConnections>
    <Connection Source="DigitalInput_I1.IN" Destination="AX_X_TO_BOOL.AX_IN"/>
    <Connection Source="AX_BOOL_TO_X.AX_OUT" Destination="DigitalOutput_Q1.OUT"/>
</AdapterConnections>
```

Der Ablauf gestaltet sich wie folgt:

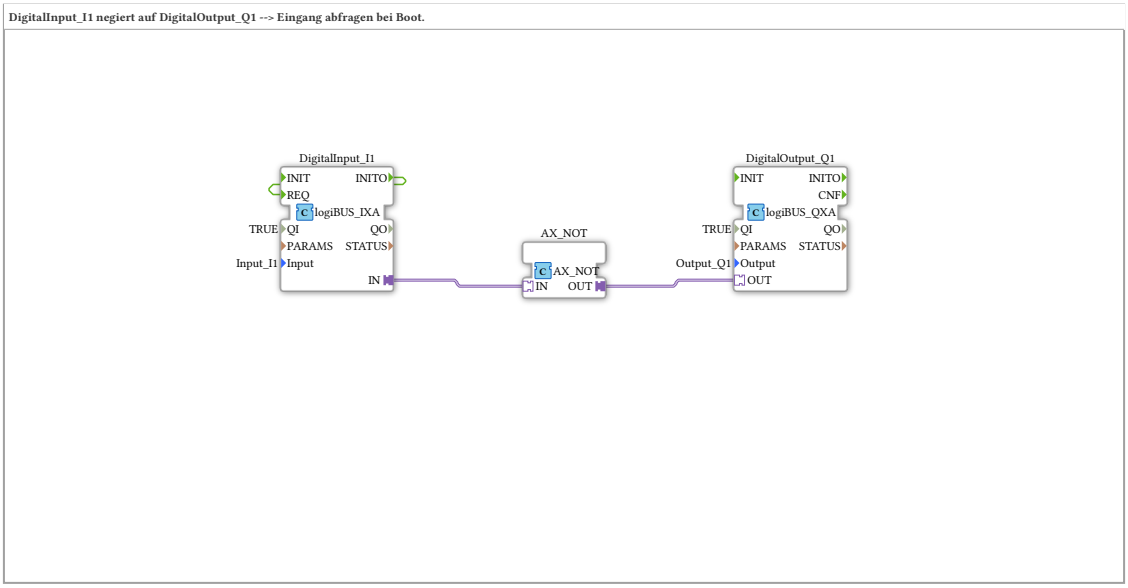
1. Ändert sich der Zustand am Eingang I1, sendet DigitalInput_I1 ein Adapter-Ereignis.
2. Der Baustein AX_X_TO_BOOL empfängt dieses, gibt den aktuellen Zustand am Datenausgang IN aus und signalisiert dies durch das Ereignis CNF.
3. Das Ereignis CNF triggert den REQ-Eingang von AX_BOOL_TO_X, welcher daraufhin den Wert von OUT übernimmt.
4. AX_BOOL_TO_X sendet ein neues Adapter-Ereignis an DigitalOutput_Q1, welcher schließlich den Ausgang Q1 schaltet.

Anwendungsbeispiel

Diese Konfiguration dient als Vorbereitung für komplexere Steuerungsaufgaben. Ein praktisches Beispiel wäre die **Invertierung eines Signals**:

Möchte man, dass die Lampe an Q1 leuchtet, wenn der Schalter an I1 *nicht* betätigt ist, kann man zwischen den Wandler-Bausteinen einfach einen NOT-Funktionsbaustein einfügen. Das boolesche Signal von AX_X_TO_BOOL.IN wird invertiert und dann an AX_BOOL_TO_X.OUT übergeben. Dies zeigt die Flexibilität, die durch die Wandlung von Adaptern in elementare Datentypen gewonnen wird.

2.1.2.6 Uebung_001c2_AX: DigitalInput_I1 negiert auf DigitalOutput_Q1 –> Eingang abfragen bei Boot



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung demonstriert die Negation eines digitalen Eingangssignals auf einen digitalen Ausgang mithilfe eines logischen Negationsadapters. Ein besonderer Fokus liegt auf dem initialen Verhalten nach dem Booten der Steuerung: durch eine Event-Rückkopplung wird sichergestellt, dass der Ausgang sofort den korrekten negierten Zustand des Eingangs annimmt.

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_I1** (Typ: `logiBUS::io::DI::logiBUS_IXA`): Liest den physischen digitalen Eingang `Input_I1` aus. *Parameter*: `QI = TRUE` (Initialisierung aktiv), `Input = "Input_I1"`.
- **DigitalOutput_Q1** (Typ: `logiBUS::io::DQ::logiBUS_QXA`): Setzt den physischen digitalen Ausgang `Output_Q1`. *Parameter*: `QI = TRUE`, `Output = "Output_Q1"`.
- **AX_NOT** (Typ: `adapter::booleanOperators::AX_NOT`): Ein Adapter-Funktionsbaustein, der den an seinem `IN`-Adapter anliegenden Booleschen Wert negiert und am `OUT`-Adapter ausgibt.

Programmablauf und Verbindungen

Der Datenfluss ist auf Adapterebene realisiert:

1. Der digitale Eingang `DigitalInput_I1` liefert den Status des physischen Eingangs als Adaptersignal an seinem `OUT`-Anschluss.
2. Dieses Signal wird direkt an den `IN`-Adapter des Negationsbausteins `AX_NOT` weitergeleitet.
3. Der negierte Wert verlässt `AX_NOT` über den `OUT`-Adapter und wird an den `OUT`-Adapter des Ausgangsbausteins `DigitalOutput_Q1` übergeben.
4. `DigitalOutput_Q1` setzt den physischen Ausgang entsprechend.

Besonderheit – Initialisierungsverhalten (Boot): Eine Eventverbindung zwischen dem Ereignisausgang `INITO` von `DigitalInput_I1` und dem Ereigniseingang `REQ` desselben Bausteins sorgt dafür, dass der Eingang sofort nach der Initialisierung (beim Booten) einmal abgefragt wird. Ohne diese Verbindung wäre der Ausgang nach dem Start zunächst `FALSE`, da die Ereigniskette erst durch ein externes Ereignis ausgelöst werden müsste. Mit der Rückkopplung wird der aktuelle Eingangswert gelesen und der Ausgang korrekt gesetzt.

Lernziele:

- Verständnis der Ereignissteuerung in 4diac (Ereignisrückkopplung zur Initialisierung).
- Anwendung von Adapterbausteinen zur Signalverarbeitung (Negation).
- Einfaches Zusammenspiel von digitalen Ein- und Ausgängen.

Zusammenfassung

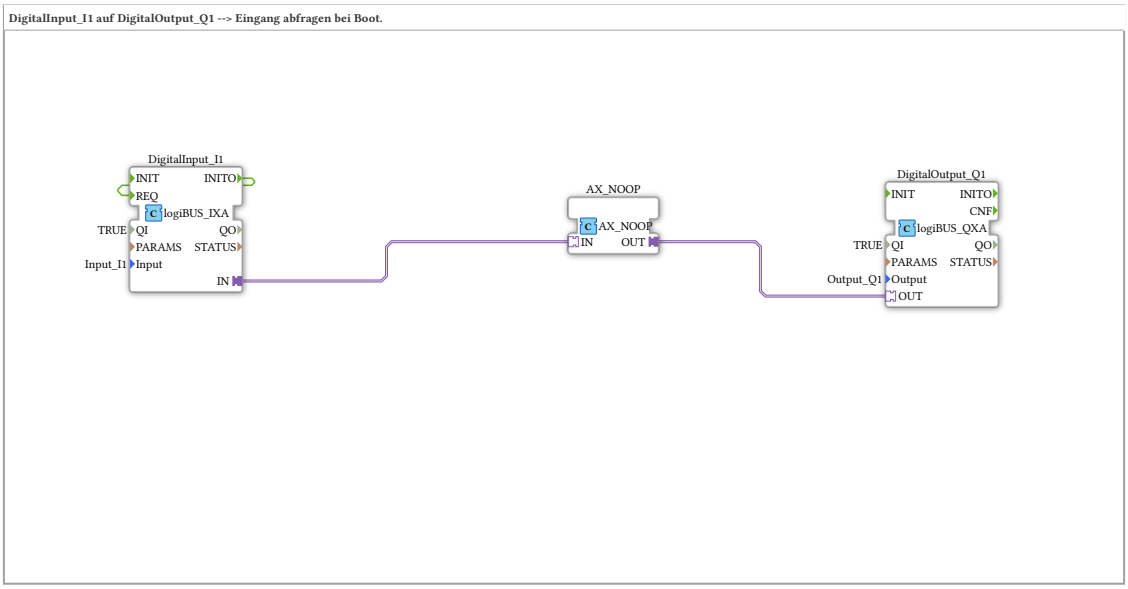
Die Übung `Uebung_001c2_AX` zeigt eine grundlegende Schaltung zur Negation eines digitalen Eingangs auf einen Ausgang. Durch die geschickte Nutzung einer Ereignisrückkopplung wird der Ausgang bereits beim Start der Steuerung mit dem korrekten Wert belegt, was die Robustheit der Applikation erhöht. Die verwendeten Bausteine (digitaler Eingang, Negationsadapter, digitaler Ausgang) sind typische Komponenten der logiBUS-Bibliothek und lassen sich flexibel in komplexere Automatisierungslösungen einbetten.

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

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

2.1.2.7 Uebung_001c3_AX: DigitalInput_I1 auf DigitalOutput_Q1 –> Eingang abfragen bei Boot

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_001c3_AX (DigitalInput_I1 auf DigitalOutput_Q1 –> Eingang abfragen bei Boot.).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **DigitalInput_I1 auf DigitalOutput_Q1 –> Eingang abfragen bei Boot.**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_001c3_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
- **DigitalInput_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
- **AX_NOOP**: Instanz des Typs adapter::booleanOperators::AX_NOOP.

Verbindungen und Schnittstellen

Adapterverbindungen:

- DigitalInput_I1.IN -> AX_NOOP.IN
- AX_NOOP.OUT -> DigitalOutput_Q1.OUT

Ereignisverbindungen:

- DigitalInput_I1.INITO -> DigitalInput_I1.REQ

Hinweise aus dem Modell

ohne die Linie INITO->REQ ist der Ausgang Q1 beim Start FALSE. mit der Linie ist er TRUE. In Adapter-Architektur entspricht der doppelten Negierung (NOT an NOT) der Baustein AX_NOOP.

Programmablauf und Funktionsweise

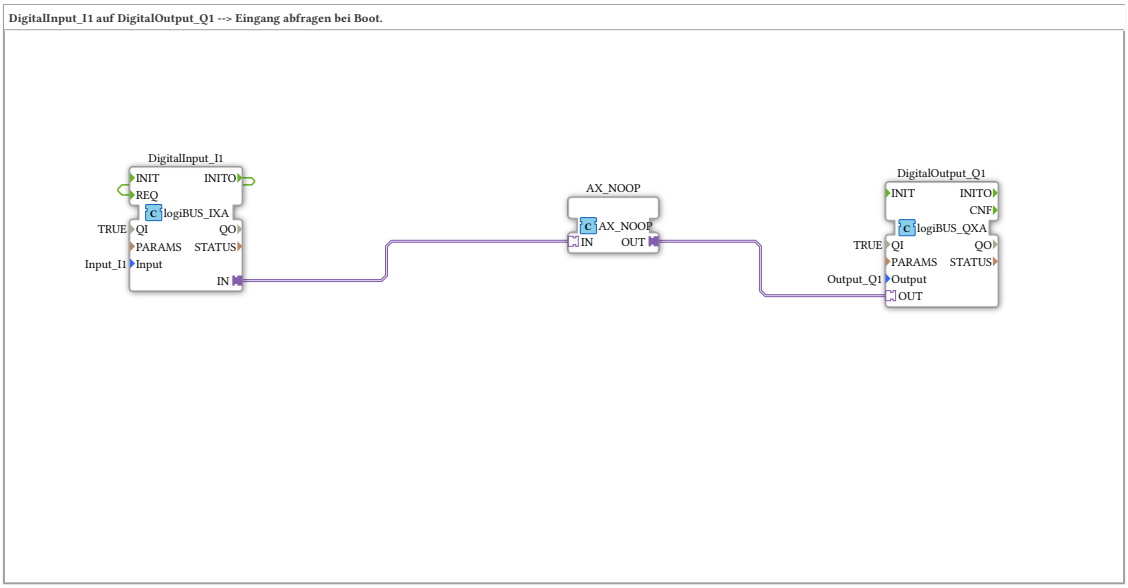
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_001c3_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.8 Uebung_001c4_AX: DigitalInput_I1 auf DigitalOutput_Q1 –> Eingang abfragen bei Boot

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_001c4_AX (DigitalInput_I1 auf DigitalOutput_Q1 –> Eingang abfragen bei Boot.).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **DigitalInput_I1 auf DigitalOutput_Q1 –> Eingang abfragen bei Boot.**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_001c4_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1

- **DigitalInput_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
- **AX_NOOP**: Instanz des Typs adapter::booleanOperators::AX_NOOP.

Verbindungen und Schnittstellen

Adapterverbindungen:

- DigitalInput_I1.IN -> AX_NOOP.IN
- AX_NOOP.OUT -> DigitalOutput_Q1.OUT

Ereignisverbindungen:

- DigitalInput_I1.INITO -> DigitalInput_I1.REQ

Hinweise aus dem Modell

ohne die Linie INITO->REQ ist der Ausgang Q1 beim Start FALSE. mit der Linie ist er TRUE. Der AX_NOOP Baustein ist die Adapter-Variante von NOOP.

Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_001c4_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.9 Uebung_001c_AX: DigitalInput_I1 auf DigitalOutput_Q1 –> Eingang abfragen bei Boot



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_001c_AX`. Hier wird demonstriert, wie ein digitaler Eingang unmittelbar nach dem Systemstart (Boot-Vorgang) abgefragt wird, um den initialen Zustand an einen digitalen Ausgang zu übertragen.

Ziel der Übung

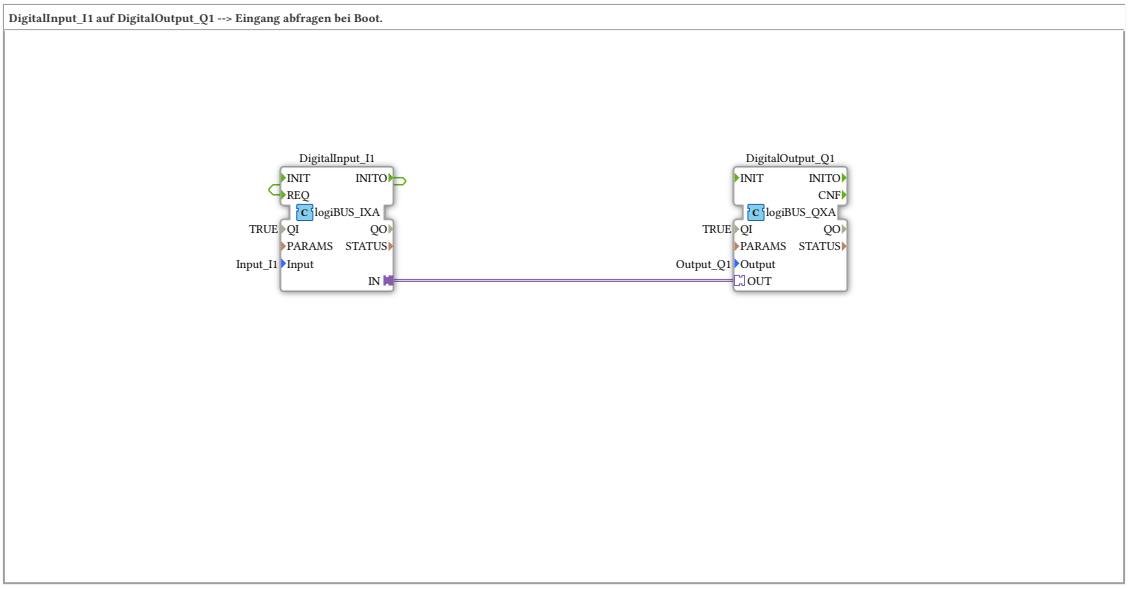
Das Hauptziel dieser Übung ist das Verständnis des Initialisierungsvorgangs in der IEC 61499. In vielen Automatisierungsszenarien reicht es nicht aus, nur auf Zustandsänderungen zu reagieren; das System muss auch beim Start den aktuellen Ist-Zustand der Hardware erfassen, um eine definierte Ausgangslage zu gewährleisten.

Beschreibung und Komponenten

Die Übung nutzt die Subapplikation `Uebung_001c_AX.SUB`, um eine Verbindung zwischen einem digitalen Eingang und einem Ausgang herzustellen, ergänzt um eine Selbst-Triggerung für den Systemstart.

Funktionsbausteine (FBs)

In der Subapplikation werden zwei zentrale Bausteine verwendet:



- **`DigitalInput_I1`**: Eine Instanz des Typs `logiBUS_IXA`. Zusätzlich zur Standardfunktion wird hier der Ereignisausgang `INITO` (Initialization Output) genutzt, um eine einmalige Abfrage beim Start auszulösen.
- **`DigitalOutput_Q1`**: Eine Instanz des Typs `logiBUS_QXA`. Dieser Baustein empfängt den initial abgefragten Wert über die Adapterverbindung und setzt den Ausgang `Output_Q1` entsprechend.

Adapter-Schnittstelle: `AX.adp`

Die Kommunikation zwischen den Bausteinen erfolgt über den bekannten Adapter-Typ `AX`, der das Ereignis `E1` und den Datenwert `D1` überträgt.

Funktionsweise

Die Besonderheit dieser Übung liegt in der Ereignisverbindung, die eine Rückkopplung für den Initialisierungsprozess schafft. Der Aufbau in der Datei `Uebung_001c_AX.SUB` sieht wie folgt aus:

```
<EventConnections>
    <Connection Source="DigitalInput_I1.INITO" Destination="DigitalInput_I1.REQ"/>
</EventConnections>
<AdapterConnections>
    <Connection Source="DigitalInput_I1.IN" Destination="DigitalOutput_Q1.OUT"/>
</AdapterConnections>
```

Der funktionale Ablauf ist wie folgt:

1. **Systemstart**: Beim Hochfahren der 4diac-Laufzeitumgebung wird der Baustein `DigitalInput_I1` initialisiert.
2. **Initialisierungs-Event**: Nach erfolgreicher Initialisierung sendet der Baustein ein `INITO`-Ereignis aus.
3. **Selbst-Triggerung**: Da `INITO` mit dem eigenen `REQ`-Eingang verbunden ist, wird der Baustein sofort aufgefordert, den physischen Zustand des Eingangs `Input_I1` zu lesen.
4. **Signalweiterleitung**: Der gelesene Wert wird über den Adapter `IN` an `DigitalOutput_Q1` gesendet, welcher den Ausgang `Q1` bereits beim Booten auf den korrekten Stand bringt.

Ohne diese `INITO -> REQ` Verbindung würde der Ausgang erst dann aktualisiert, wenn sich der Zustand des Eingangs zum ersten Mal *nach* dem Start ändert.

Anwendungsbeispiel

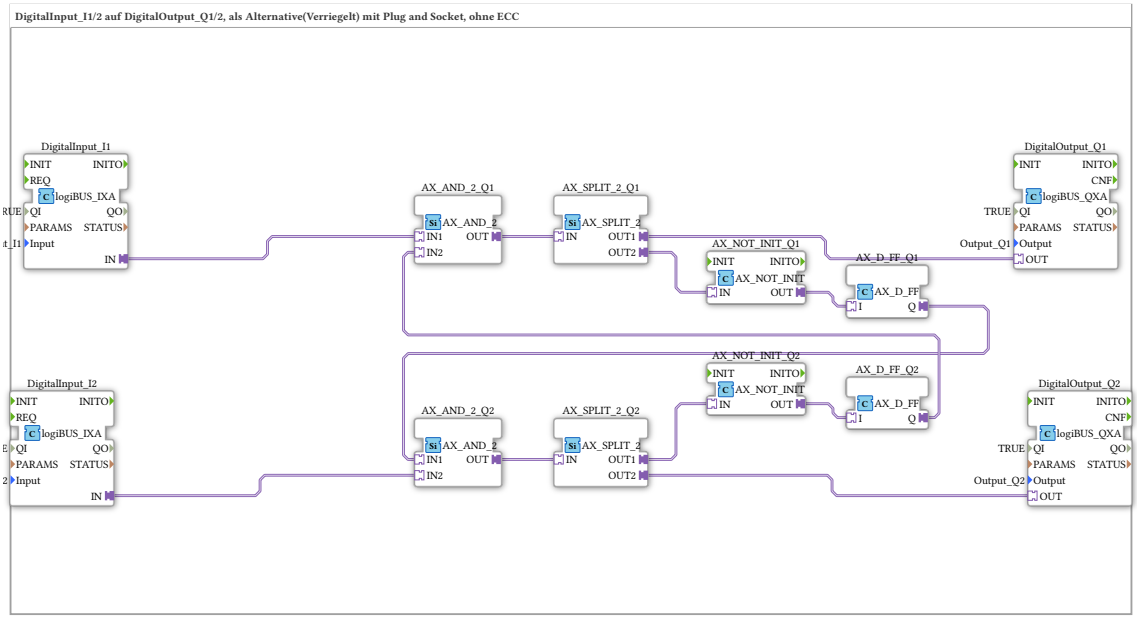
Ein praktisches Beispiel ist die **Zustandssynchronisation nach einem Stromausfall**:

Stellen Sie sich vor, eine Steuerung steuert eine Lüftungsklappe basierend auf der Position eines Schalters. Wenn die Steuerung neu startet, muss sie sofort wissen, in welcher Position der Schalter steht, um die Klappe korrekt anzusteuern, noch bevor der Bediener den Schalter erneut betätigt. Die Boot-Abfrage stellt sicher, dass Software-Zustand und Hardware-Realität von der ersten Sekunde an synchron sind.

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

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

2.1.2.10 Uebung_001d2_AX: DigitalInput_I1/2 auf DigitalOutput_Q1/2, als Alternative(Verriegelt) mit Plug and Socket, ohne ECC



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung realisiert eine alternative, verschaltete Ansteuerung zweier Digitalausgänge (Q1, Q2) durch zwei Digitaleingänge (I1, I2). Die Schaltung bildet eine verriegelte (gegenseitig ausschließende) Zuordnung mit Hilfe von Logikgattern und Flip-Flops, jedoch ohne Verwendung eines expliziten ECC (Execution Control Chart). Sie dient als Beispiel für eine komplexere Kopplung von Eingangs- und Ausgangssignalen basierend auf dem logiBUS-IO-System.

Die Schaltung verarbeitet die Eingänge so, dass immer nur einer der beiden Ausgänge aktiv werden kann. Welcher Eingang den Ausgang steuert, wird durch die interne Logik festgelegt, wobei die Rückkopplung der Flip-Flop-Zustände eine wechselseitige Sperre realisiert.

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_I1**, **DigitalInput_I2**: Typ: `logiBUS::io::DI::logiBUS_IXA` Parametrisierung:
  - `QI` = TRUE (aktiviert)
  - `Input` = `Input_I1` bzw. `Input_I2` (logiBUS-Kanal) Diese Bausteine stellen die physikalischen Digitaleingänge (z. B. von einem Taster oder Sensor) als boolesche Signale im System bereit.
- **DigitalOutput_Q1**, **DigitalOutput_Q2**: Typ: `logiBUS::io::DQ::logiBUS_QXA` Parametrisierung:
  - `QI` = TRUE
  - `Output` = `Output_Q1` bzw. `Output_Q2` Sie steuern die physikalischen Digitalausgänge (z. B. Lampen oder Relais).
- **AX_AND_2_Q1**, **AX_AND_2_Q2**: Typ: `adapter::booleanOperators::AX_AND_2` Parameter: keine (Standardkonfiguration) Realisiert eine logische UND-Verknüpfung mit zwei Eingängen (IN1, IN2) und einem Ausgang (OUT).
- **AX_SPLIT_2_Q1**, **AX_SPLIT_2_Q2**: Typ: `adapter::events::unidirectional::AX_SPLIT_2` Ein Ereignis-Splitter, der ein eingehendes Ereignis (IN) auf zwei Ausgänge (OUT1, OUT2) verteilt. Dient zur gleichzeitigen Bereitstellung eines Signals für zwei nachfolgende Bausteine.

- **AX_NOT_INIT_Q1**, **AX_NOT_INIT_Q2**: Typ: `adapter::booleanOperators::AX_NOT_INIT` Eine logische Negation (NOT), deren Ausgangszustand beim Start initialisiert werden muss. Wandelt ein logisches Signal in sein Gegenteil um.
- **AX_D_FF_Q1**, **AX_D_FF_Q2**: Typ: `adapter::events::unidirectional::AX_D_FF` Ein D-Flip-Flop (Data Flip-Flop). Es speichert den Wert am Dateneingang (I) bei einem aktiven Takt-Ereignis und gibt ihn am Ausgang (Q) aus. Verwendet zur Zustandsspeicherung innerhalb der Verriegelung.

Programmablauf und Verbindungen

Die Schaltung arbeitet nach folgendem Prinzip:

1. **Signalaufbereitung** Die Eingänge I1 und I2 werden über die `logiBUS_IXA`-Bausteine eingelesen und als boolesche Signale an die nachfolgende Logik weitergegeben.
2. **UND-Verknüpfung mit Rückkopplung**
   - Der Ausgang von DigitalInput_I1 wird auf den ersten Eingang (IN1) von `AX_AND_2_Q1` geführt.
   - Der Ausgang von DigitalInput_I2 wird auf den zweiten Eingang (IN2) von `AX_AND_2_Q2` geführt.
   - Die UND-Gatter erhalten jeweils den zweiten Eingang aus dem *Ausgang des anderen* D-Flip-Flops:
     - `AX_AND_2_Q1.IN2` ist mit dem Ausgang `Q` von `AX_D_FF_Q2` verbunden.
     - `AX_AND_2_Q2.IN1` ist mit dem Ausgang `Q` von `AX_D_FF_Q1` verbunden.

   Diese Kreuzverschaltung bewirkt: Ausgang Q1 kann nur aktiv werden, wenn das Flip-Flop Q2 nicht aktiv ist (und umgekehrt). Damit wird eine gegenseitige Verriegelung realisiert.
3. **Signalverteilung und Negation**
   - Die Ausgänge der UND-Gatter werden durch die `AX_SPLIT_2`-Bausteine auf zwei Pfade aufgeteilt:
     - Ein Pfad geht direkt zu den Digitalausgängen: `AX_SPLIT_2_Q1.OUT1` ⟶ `DigitalOutput_Q1.OUT` `AX_SPLIT_2_Q2.OUT2` ⟶ `DigitalOutput_Q2.OUT`
     - Der andere Pfad geht über die Negation (`AX_NOT_INIT`) zu den D-Flip-Flops: `AX_SPLIT_2_Q1.OUT2` ⟶ `AX_NOT_INIT_Q1.IN` ⟶ `AX_D_FF_Q1.I` `AX_SPLIT_2_Q2.OUT1` ⟶ `AX_NOT_INIT_Q2.IN` ⟶ `AX_D_FF_Q2.I`

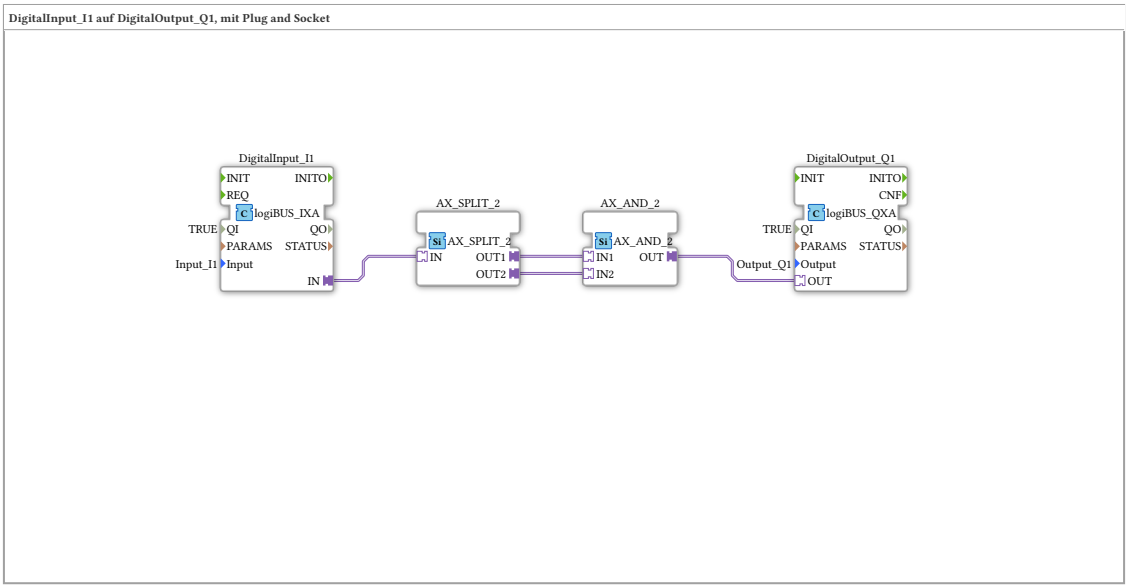
   Die Negation stellt sicher, dass das Flip-Flop beim nächsten Takt den invertierten Wert des UND-Ausgangs speichert, sodass eine wechselseitige Abstimmung möglich wird.
4. **Zustandsspeicherung** Die D-Flip-Flops speichern den negierten Zustand der jeweiligen UND-Verknüpfung und stellen ihn als Rückmeldesignal für die UND-Gatter der anderen Seite bereit.
5. **Ergebnis** Insgesamt ergibt sich ein System, bei dem nur einer der beiden Ausgänge (Q1 oder Q2) gleichzeitig aktiv sein kann. Ein Wechsel des aktiven Ausgangs erfolgt, sobald der zugehörige Eingang gesetzt und der andere Eingang zurückgesetzt wird – die interne Logik sorgt für die Umschaltung durch die Flip-Flop-Zustände.

Zusammenfassung

Die Übung zeigt eine komplexere Lösung für die verriegelte Ansteuerung von zwei Ausgängen durch zwei Eingänge unter Verwendung von UND-Gattern, Splittern, Negatoren und D-Flip-Flops. Sie demonstriert, wie mit einfachen Logikbausteinen eine gegenseitige Ausschließlichkeit realisiert werden kann, ohne auf einen dedizierten Verriegelungsbaustein (wie ILOCK) zurückzugreifen.

Der Kommentar im Quellcode (`Kompliziert. BESSER: ILOCK-Baustein`) weist darauf hin, dass ein spezialisierter ILOCK-Funktionsbaustein diese Aufgabe wesentlich einfacher lösen würde. Die Übung eignet sich daher gut, um die Funktionsweise von Flip-Flop-basierten Verriegelungen zu verstehen sowie die händische Implementierung mit Standardbausteinen zu erlernen. Sie setzt Grundkenntnisse der logiBUS-IO-Anbindung und der booleschen Logik voraus.

2.1.2.11 Uebung_001d_AX: DigitalInput_I1 auf DigitalOutput_Q1, mit Plug and Socket



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung demonstriert das Durchschalten eines digitalen Eingangssignals (Input_I1) auf einen digitalen Ausgang (Output_Q1). Dabei kommen **Adapterbausteine** (Plug-and-Socket) zum Einsatz, um die Ereignis- und Datenflüsse zwischen den Funktionsbausteinen zu koppeln. Das Ziel ist es, die grundlegende Verwendung von Adapterverbindungen in der 4diac-IDE zu verstehen.

Verwendete Funktionsbausteine (FBs)

DigitalInput_I1

- **Typ**: `logiBUS::io::DI::logiBUS_IXA`
- **Verwendete Parameter**:
  - `QI` = `TRUE` (Freigabe aktiv)
  - `Input` = `Input_I1` (physischer Eingang)
- **Funktionsweise**: Der Baustein liest den Zustand des angeschlossenen digitalen Eingangs **Input_I1**. Bei einer Signaländerung wird ein Ereignis über den **Adapterausgang `IN`** ausgegeben. Der Parameter `QI` muss gesetzt sein, damit der Baustein arbeitet.

DigitalOutput_Q1

- **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
- **Verwendete Parameter**:
  - `QI` = `TRUE` (Freigabe aktiv)
  - `Output` = `Output_Q1` (physischer Ausgang)
- **Funktionsweise**: Der Baustein empfängt ein Ereignis über den **Adaptereingang `OUT`** und setzt den angeschlossenen digitalen Ausgang **Output_Q1** entsprechend. Der Ausgang wird aktiv, sobald ein Ereignis eintrifft.

AX_SPLIT_2

- **Typ**: `adapter::events::unidirectional::AX_SPLIT_2`
- **Funktionsweise**: Dieser Adapterbaustein verteilt ein eingehendes Ereignis auf **zwei identische Ausgänge** (`OUT1` und `OUT2`). Er dient als **Splitte**r, um das Signal parallel an mehrere nachfolgende Bausteine weiterzuleiten.

AX_AND_2

- **Typ**: `adapter::booleanOperators::AX_AND_2`

- **Funktionsweise**: Dieser Adapterbaustein führt eine **logische UND-Verknüpfung** auf zwei Ereigniseingängen (`IN1`, `IN2`) durch. Nur wenn an beiden Eingängen gleichzeitig ein Ereignis anliegt, wird ein Ereignis am Ausgang `OUT` ausgegeben.

Programmablauf und Verbindungen

Der Signalfluss in der Sub-Applikation erfolgt ausschließlich über **Adapterverbindungen** (Plug and Socket):

1. Der Digitaleingang **DigitalInput_I1** erkennt eine Änderung an `Input_I1` und sendet ein Ereignis auf seinem Adapterausgang `IN`.
2. Dieses Ereignis wird zum **AX_SPLIT_2**-Baustein geleitet, der es auf seine beiden Ausgänge `OUT1` und `OUT2` vervielfältigt.
3. Beide Ausgänge sind mit den Eingängen des **AX_AND_2**-Bausteins verbunden (`IN1` und `IN2`). Da beide Eingänge dasselbe Ereignis gleichzeitig erhalten, führt die UND-Verknüpfung immer zu einem Ereignis am Ausgang `OUT`.
4. Das Ausgangsereignis von `AX_AND_2` wird an den Adaptereingang `OUT` des **DigitalOutput_Q1**-Bausteins übertragen, der daraufhin den physischen Ausgang **Output_Q1** setzt.

Im Ergebnis wird der Digitaleingang **I1** direkt auf den Digitalausgang **Q1** abgebildet. Die Zwischenschaltung von `AX_SPLIT_2` und `AX_AND_2` dient lediglich der Demonstration von Adapterverknüpfungen und hat keine logische Auswirkung auf das Durchschaltverhalten.

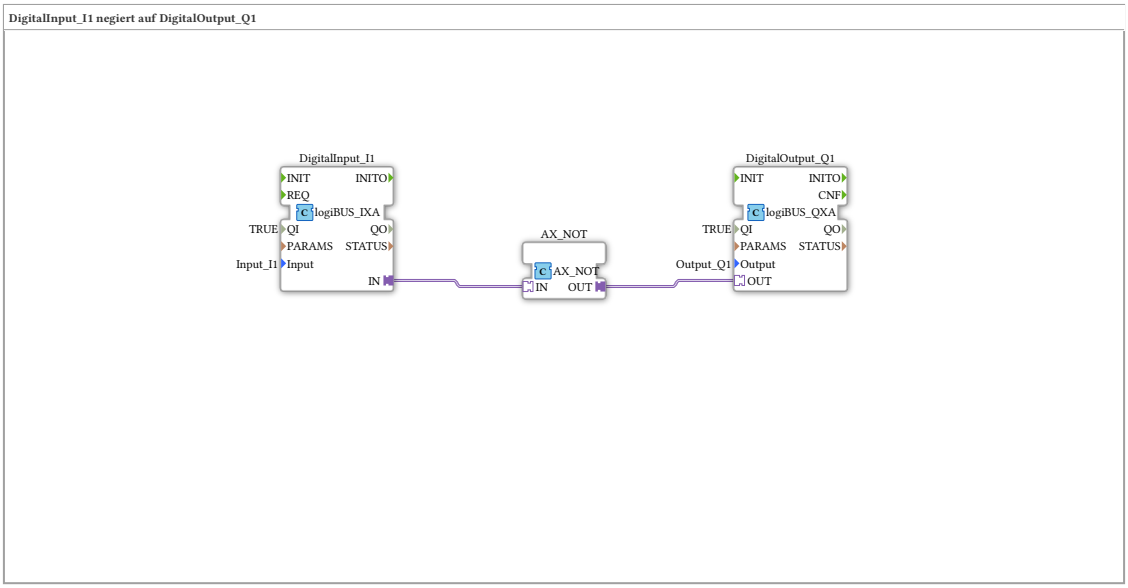
Zusammenfassung

Die Übung `Uebung_001d_AX` zeigt, wie man mit **Plug-and-Socket-Verbindungen** (Adapterbausteine) Ereignisse zwischen Funktionsbausteinen koppelt, ohne direkte Datenleitungen zu verwenden. Durch die Kombination von Splitter- (`AX_SPLIT_2`) und UND-Baustein (`AX_AND_2`) wird ein einfaches Durchschalten realisiert. Dies vermittelt ein grundlegendes Verständnis für die ereignisgesteuerte Kommunikation in der 4diac-IDE und die Verwendung von Adaptern in der Automatisierungstechnik.

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

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

2.1.2.12 Uebung_001e_AX: DigitalInput_I1 negiert auf DigitalOutput_Q1



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung realisiert eine einfache boolesche Negation: Der Zustand des digitalen Eingangs **Input_I1** wird negiert und auf den digitalen Ausgang **Output_Q1** ausgegeben. Sie dient als Einstieg in die Signal-

verarbeitung mit 4diac und zeigt die grundlegende Verschaltung eines Eingangsmoduls, eines logischen Negationsbausteins und eines Ausgangsmoduls über Adapterverbindungen.

Verwendete Funktionsbausteine (FBs)

Die Übung verwendet drei konkrete Funktionsbausteine aus der Bibliothek:

- **DigitalInput_I1**
  - **Typ**: `logiBUS::io::DI::logiBUS_IXA`
  - **Parameter**:
    - `QI` = `TRUE` (Eingang aktiviert)
    - `Input` = `Input_I1` (physikalischer Eingang)
  - **Funktion**: Liest den digitalen Zustand des angeschlossenen Sensors (z. B. Taster oder Schalter) am Eingang I1.
- **DigitalOutput_Q1**
  - **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
  - **Parameter**:
    - `QI` = `TRUE` (Ausgang aktiviert)
    - `Output` = `Output_Q1` (physikalischer Ausgang)
  - **Funktion**: Gibt den empfangenen logischen Wert auf den digitalen Ausgang Q1 aus (z. B. zur Ansteuerung einer LED).
- **AX_NOT**
  - **Typ**: `adapter::booleanOperators::AX_NOT`
  - **Parameter**: Keine
  - **Funktion**: Führt eine boolesche Negation (NOT-Operation) auf den eingehenden Adapter-Datenwert durch. Der Ausgang `OUT` liefert den invertierten Wert des Eingangs `IN`.

Programmablauf und Verbindungen

Der Datenfluss erfolgt über drei **Adapter-Verbindungen** (im XML als `<AdapterConnections>` definiert):

1. `DigitalInput_I1.IN` ⟶ `AX_NOT.IN` Der gelesene Zustand des Eingangs I1 wird an den Negationsbaustein weitergeleitet.
2. `AX_NOT.OUT` ⟶ `DigitalOutput_Q1.OUT` Der negierte Wert wird an den Ausgangsbaustein übergeben.

Dadurch ergibt sich folgende Logik:

- Wenn **Input_I1** `TRUE` (z. B. Taster gedrückt) ⟶ `AX_NOT` liefert `FALSE` ⟶ **Output_Q1** wird `FALSE` (LED aus).
- Wenn **Input_I1** `FALSE` (Taster nicht gedrückt) ⟶ `AX_NOT` liefert `TRUE` ⟶ **Output_Q1** wird `TRUE` (LED an).

Die Parameter `QI = TRUE` bei Ein- und Ausgangsbaustein aktivieren diese dauerhaft. Es sind keine weiteren Verschaltungen oder Ereignissteuerungen erforderlich.

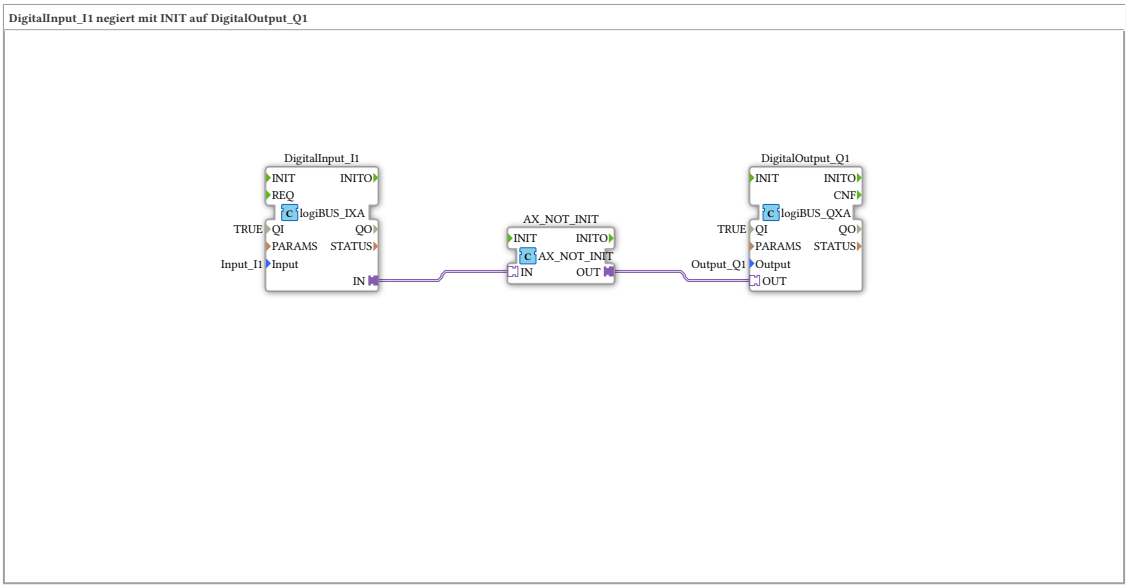
Zusammenfassung

Die Übung **Uebung_001e_AX** vermittelt die Grundlagen der Adapter-basierten Kommunikation in 4diac. Sie zeigt, wie ein digitales Eingangssignal mit einem einfachen Logikbaustein (NOT) verarbeitet und auf einen physikalischen Ausgang gelegt wird. Dieses Verständnis ist die Basis für komplexere Verknüpfungen und Steuerungsaufgaben in der Automatisierungstechnik.

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

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

2.1.2.13 Uebung_001f_AX: DigitalInput_I1 negiert mit INIT auf DigitalOutput_Q1



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung demonstriert die Negation eines digitalen Eingangssignals unter Verwendung des Funktionsbausteins `AX_NOT_INIT`. Das negierte Signal wird auf einen digitalen Ausgang gegeben. Ein besonderer Effekt tritt beim Start (BOOT) auf: Da der Eingang I1 beim Systemstart nicht abgefragt wird, liefert `AX_NOT_INIT` zu Beginn einen `TRUE`-Wert, unabhängig vom tatsächlichen Eingangszustand.

Verwendete Funktionsbausteine (FBs)

Die Übung setzt sich aus drei Funktionsbausteinen zusammen, die im Netzwerk der SubApp miteinander verbunden sind.

Sub-Bausteine: DigitalInput_I1

- **Typ**: `logiBUS::io::DI::logiBUS_IXA`
- **Verwendete interne FBs**: keine
- **Parameter**:
  - `QI` = TRUE
  - `Input` = `Input_I1`
- **Ereignisausgang/-eingang**: keine (rein datengetrieben über Adapter)
- **Datenausgang/-eingang**: `IN` (Adapter-Ausgang, liefert den digitalen Eingangswert)
- **Funktionsweise**: Der Baustein liest den Zustand des digitalen Eingangs `Input_I1` und stellt ihn über den Adapterausgang `IN` bereit.

Sub-Bausteine: DigitalOutput_Q1

- **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
- **Verwendete interne FBs**: keine
- **Parameter**:
  - `QI` = TRUE
  - `Output` = `Output_Q1`
- **Ereignisausgang/-eingang**: keine
- **Datenausgang/-eingang**: `OUT` (Adapter-Eingang, nimmt den zu setzenden Ausgangswert entgegen)
- **Funktionsweise**: Der Baustein setzt den digitalen Ausgang `Output_Q1` auf den Wert, der am Adaptereingang `OUT` anliegt.

Sub-Bausteine: AX_NOT_INIT

- **Typ**: `adapter::booleanOperators::AX_NOT_INIT`
- **Verwendete interne FBs**: keine
- **Parameter**: keine

- **Ereignisausgang/-eingang**: keine
- **Datenausgang/-eingang**:
  - `IN` (Adapter-Eingang, zu negierender Wert)
  - `OUT` (Adapter-Ausgang, negierter Wert)
- **Funktionsweise**: Der Baustein negiert den am Eingang `IN` anliegenden booleschen Wert. Bei Systemstart (BOOT) wird der Ausgang `OUT` auf `TRUE` gesetzt, selbst wenn der Eingang noch nicht gelesen wurde. Dieses Verhalten wird durch den Namenszusatz `_INIT` gekennzeichnet.

Programmablauf und Verbindungen

Die Verbindungen im SubApp-Netzwerk sind als Adapterverbindungen realisiert:

1. Der Adapterausgang `IN` von `DigitalInput_I1` wird mit dem Adaptereingang `IN` von `AX_NOT_INIT` verbunden.
2. Der Adapterausgang `OUT` von `AX_NOT_INIT` wird mit dem Adaptereingang `OUT` von `DigitalOutput_Q1` verbunden.

Ablauf:

- Der digitale Eingangswert wird ständig aktualisiert und an `AX_NOT_INIT` weitergegeben.
- `AX_NOT_INIT` negiert den Wert und gibt das Ergebnis an `DigitalOutput_Q1` weiter.
- Der Ausgangsbaustein setzt den physikalischen Ausgang entsprechend.

Besonderheit beim Start: Während der Initialisierungsphase (BOOT) hat `AX_NOT_INIT` noch keinen gültigen Eingangswert empfangen. Daher gibt er seinen vordefinierten Startwert `TRUE` aus. Dies führt dazu, dass der Ausgang kurzzeitig `TRUE` wird, auch wenn der Eingang eigentlich `FALSE` ist.

Lernziele:

- Verständnis der Negation boolescher Signale in 4diac.
- Kennenlernen des Startverhaltens initialisierter Funktionsbausteine (`INIT`-Bausteine).
- Umgang mit Adapterverbindungen zwischen einzelnen Funktionsbausteinen.

Schwierigkeitsgrad: Einfach – geeignet für Einsteiger, die grundlegende Signalverarbeitung und das Verhalten von Bausteinen in 4diac nachvollziehen möchten.

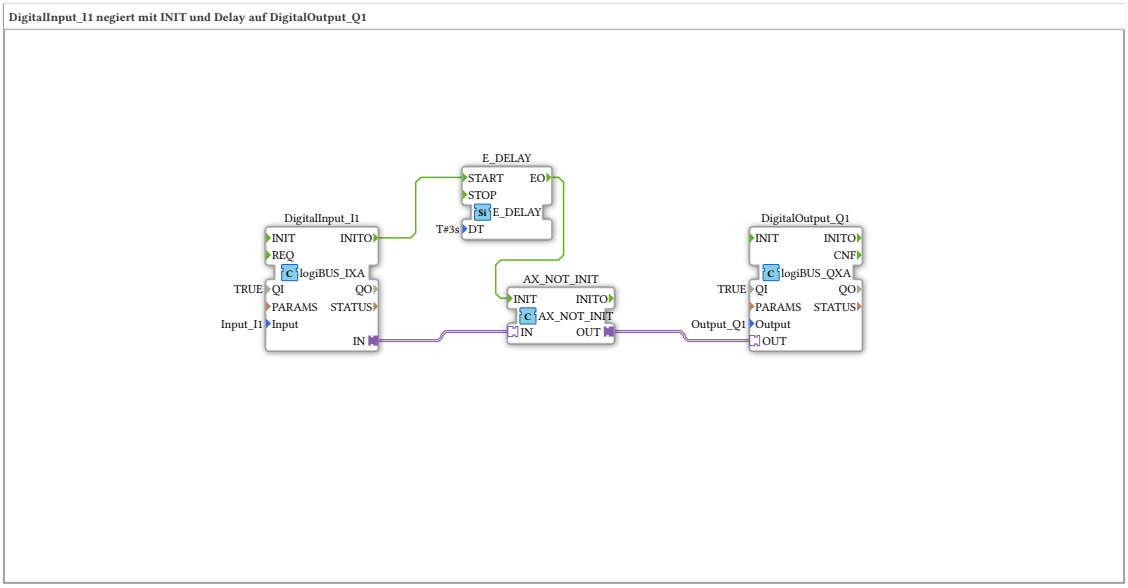
Zusammenfassung

Die Übung `Uebung_001f_AX` veranschaulicht die Negation eines digitalen Eingangssignals unter Verwendung des speziellen Bausteins `AX_NOT_INIT`. Die Besonderheit liegt im initialen Ausgangszustand beim Systemstart, der unabhängig vom Eingang `TRUE` ist. Die Übung wird mit dem Kommentar ergänzt: *„obwohl I1 nicht abgefragt wird beim BOOT, wird AX_NOT hier TRUE ausgeben."* Damit wird der Lerneffekt zum Startverhalten initialisierter Bausteine unterstrichen.

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

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

2.1.2.14 Uebung_001g_AX: DigitalInput_I1 negiert mit INIT und Delay auf DigitalOutput_Q1



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung demonstriert die Verarbeitung eines digitalen Eingangssignals (I1) unter Verwendung einer Negation und einer zeitlichen Verzögerung. Der Eingangswert wird nach einem initialen Ereignis und einer definierten Verzögerung negiert auf einen digitalen Ausgang (Q1) ausgegeben. Ein besonderer Fokus liegt auf dem Verhalten des Negationsbausteins, der auch dann einen gültigen Wert (TRUE) liefert, wenn der Eingang beim Systemstart noch nicht abgefragt wurde.

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: `DigitalInput_I1`

- **Typ**: `logiBUS::io::DI::logiBUS_IXA`
- **Verwendete interne FBs**: Keine
- **Parameter**:
  - `QI` = `TRUE`
  - `Input` = `Input_I1`
- **Ereignisausgang**: `INITO` (wird beim Abschluss der Initialisierung ausgelöst)
- **Datenausgang**: `IN` (liefert den aktuellen digitalen Eingangswert)

Sub-Baustein: `AX_NOT_INIT`

- **Typ**: `adapter::booleanOperators::AX_NOT_INIT`
- **Verwendete interne FBs**: Keine
- **Ereigniseingang**: `INIT` (löst die Berechnung aus)
- **Adaptereingang**: `IN` (erwartet einen booleschen Wert über einen Adapter)
- **Adapterausgang**: `OUT` (liefert den negierten Wert des Eingangs)

Sub-Baustein: `E_DELAY`

- **Typ**: `iec61499::events::E_DELAY`
- **Verwendete interne FBs**: Keine
- **Parameter**:
  - `DT` = `T#3s` (Verzögerungszeit von 3 Sekunden)
- **Ereigniseingang**: `START` (startet den Timer)
- **Ereignisausgang**: `EO` (wird nach Ablauf der Verzögerungszeit ausgelöst)

Sub-Baustein: `DigitalOutput_Q1`

- **Typ**: `logiBUS::io::DQ::logiBUS_QXA`

- **Verwendete interne FBs**: Keine
- **Parameter**:
  - `QI` = `TRUE`
  - `Output` = `Output_Q1`
- **Adaptereingang**: `OUT` (erwartet den zu setzenden digitalen Wert)

Programmablauf und Verbindungen

Der Ablauf ist ereignisgesteuert und folgt dieser Reihenfolge:

1. **Initialisierung**: Der Baustein `DigitalInput_I1` führt beim Systemstart seine Initialisierung durch. Nach erfolgreicher Initialisierung wird das Ereignis `INITO` ausgelöst.
2. **Verzögerung starten**: Das Ereignis `INITO` wird über eine **Event-Verbindung** an den Eingang `START` des Bausteins `E_DELAY` weitergeleitet. Dieser startet einen Timer mit einer Verzögerungszeit von 3 Sekunden (`DT = T#3s`).
3. **Negation auslösen**: Nach Ablauf der 3 Sekunden sendet `E_DELAY` das Ereignis `EO` an den Eingang `INIT` des Bausteins `AX_NOT_INIT`. Dadurch wird die Negation des aktuell anliegenden Eingangswertes berechnet.
4. **Wertübergabe**: Der aktuelle digitale Eingangswert von `DigitalInput_I1` wird über eine **Adapter-Verbindung** (`IN`) an den Baustein `AX_NOT_INIT` übergeben. Dessen Ausgang `OUT` liefert den negierten Wert (`NOT`).
5. **Ausgabe**: Der negierte Wert wird über eine weitere **Adapter-Verbindung** an den Eingang `OUT` des Bausteins `DigitalOutput_Q1` übergeben und damit auf den Ausgang `Output_Q1` geschrieben.

Hinweis aus dem Kommentar: Da der Eingang `I1` beim Bootvorgang nicht sofort abgefragt wird, gibt der Baustein `AX_NOT_INIT` in der Zwischenzeit den Wert `TRUE` aus, bis der erste gültige Eingangswert verarbeitet wurde.

Lernziele und Vorkenntnisse:

- **Schwierigkeitsgrad**: Einsteiger
- **Vorkenntnisse**: Grundlegendes Verständnis der 4diac-IDE, Ereignis- und Datenverbindungen.
- **Lernziele**:
  - Umgang mit digitalen Ein- und Ausgängen.
  - Verwendung von Verzögerungsbausteinen (`E_DELAY`).
  - Anwendung von Negationsbausteinen mit Initialisierungssteuerung (`AX_NOT_INIT`).
  - Verständnis des Initialisierungsverhaltens und der Ereignisverkettung.

Start der Übung: Importieren Sie die SubApp in ein 4diac-Projekt, weisen Sie die Ein- und Ausgänge den entsprechenden Hardware- oder Simulationsressourcen zu und führen Sie die Konfiguration aus.

Zusammenfassung

Die Übung `Uebung_001g_AX` zeigt, wie ein digitales Eingangssignal nach einer definierten Verzögerung von 3 Sekunden negiert auf einen digitalen Ausgang gegeben wird. Dabei wird das Initialisierungsereignis des Eingangs genutzt, um den Timer zu starten, und der spezielle Negationsbaustein `AX_NOT_INIT` stellt sicher, dass auch beim Fehlen eines initialen Eingangswertes ein definierter Ausgangszustand (`TRUE`) ausgegeben wird. Dies ist ein grundlegendes Beispiel für die ereignisgesteuerte Signalverarbeitung in 4diac.

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

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

2.1.2.15 Uebung_002_AX: DigitalInput_I1 auf DigitalOutput_Q1/_Q2, mit Plug and Socket



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_002_AX`, bei der ein einzelnes digitales Eingangssignal auf zwei verschiedene digitale Ausgänge verteilt wird. Hierbei kommt das Konzept der Adapter-Verzweigung zum Einsatz.

Ziel der Übung

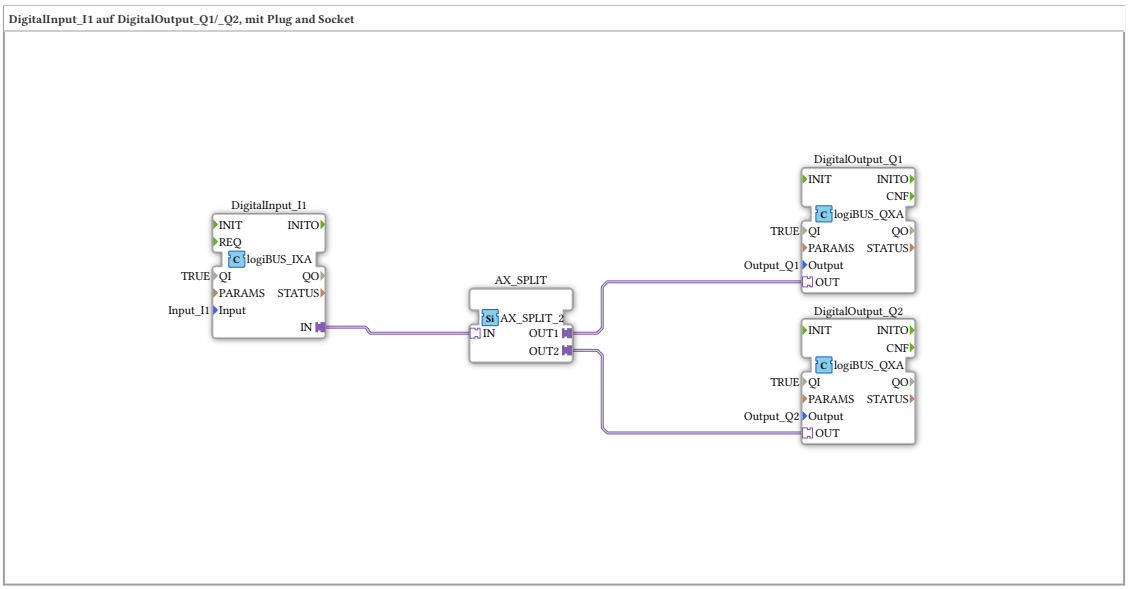
Das Hauptziel dieser Übung ist es, zu zeigen, wie Adapterverbindungen in der IEC 61499 verzweigt werden können. Da ein „Plug“ (Ausgang eines Adapters) in 4diac oft nur mit einer „Socket“ (Eingang eines Adapters) verbunden werden kann (je nach Version und Konfiguration), wird hier ein spezieller Splitter-Baustein verwendet, um ein Signal sauber auf mehrere Empfänger zu verteilen.

Beschreibung und Komponenten

In der Subapplikation `Uebung_002_AX.SUB` wird ein digitaler Eingang eingelesen und über einen Adapter-Splitter an zwei digitale Ausgänge weitergereicht.

Funktionsbausteine (FBs)

Folgende Bausteine kommen zum Einsatz:



- **`DigitalInput_I1`**: Eine Instanz des Typs `logiBUS_IXA`. Dieser Baustein liest den Hardware-Eingang `Input_I1`.
- **`AX_SPLIT`**: Eine Instanz des Typs `AX_SPLIT_2`. Dieser Baustein verfügt über einen Adapter-Eingang (`IN`) und zwei identische Adapter-Ausgänge (`OUT1`, `OUT2`) und fungiert somit als Signalvervielfältiger.
- **`DigitalOutput_Q1`** & **`DigitalOutput_Q2`**: Instanzen des Typs `logiBUS_QXA`. Diese repräsentieren die physischen Ausgänge `Output_Q1` und `Output_Q2`.

Adapter-Schnittstelle: `AX.adp`

Auch in dieser Übung wird der unidirektionale Adapter-Typ `AX` verwendet, der Ereignisse und Datenwerte für die Übertragung bündelt.

Funktionsweise

Die Signalverteilung wird durch die zentrale Position des `AX_SPLIT`-Bausteins im Netzwerk erreicht. Der Aufbau in `Uebung_002_AX.SUB` ist wie folgt definiert:

```
<AdapterConnections>
    <Connection Source="DigitalInput_I1.IN" Destination="AX_SPLIT.IN"/>
    <Connection Source="AX_SPLIT.OUT1" Destination="DigitalOutput_Q1.OUT"/>
    <Connection Source="AX_SPLIT.OUT2" Destination="DigitalOutput_Q2.OUT"/>
</AdapterConnections>
```

Der Signalweg verläuft dabei in folgenden Schritten:

1. Der Baustein `DigitalInput_I1` detektiert eine Änderung am physischen Eingang.
2. Ein Adapter-Ereignis wird an den `AX_SPLIT`-Baustein gesendet.
3. Der `AX_SPLIT`-Baustein repliziert dieses Ereignis und den dazugehörigen Datenwert (`D1`) unmittelbar an seine beiden Ausgänge `OUT1` und `OUT2`.
4. Beide Ausgangsbausteine (`DigitalOutput_Q1` und `DigitalOutput_Q2`) empfangen das Signal zeitgleich und schalten ihre jeweiligen Hardware-Ausgänge.

Im Ergebnis schalten beide Ausgänge synchron zum Zustand des Eingangs `I1`.

Anwendungsbeispiel

Ein typisches Anwendungsbeispiel ist die **parallele Statusanzeige**:

Ein Sensor an einer Maschine (`I1`) soll nicht nur die interne Logik steuern, sondern gleichzeitig eine lokale Kontrollleuchte (`Q1`) und eine Signallampe an einem entfernten Bedienpult (`Q2`) aktivieren. Durch den Einsatz des Splitters wird sichergestellt, dass beide Anzeigen immer den identischen Zustand des Sensors widerspiegeln, ohne dass die Logik für jeden Ausgang separat implementiert werden muss.

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

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

2.1.2.16 Uebung_002a2_AX: DigitalInput_I1/_I2 mit F_AND (bitwise) auf DigitalOutput_Q1

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_002a2_AX`. Hier wird gezeigt, wie Adapter-Signale in boolesche Werte gewandelt werden, um sie mit Standard-Logikbausteinen zu verarbeiten.

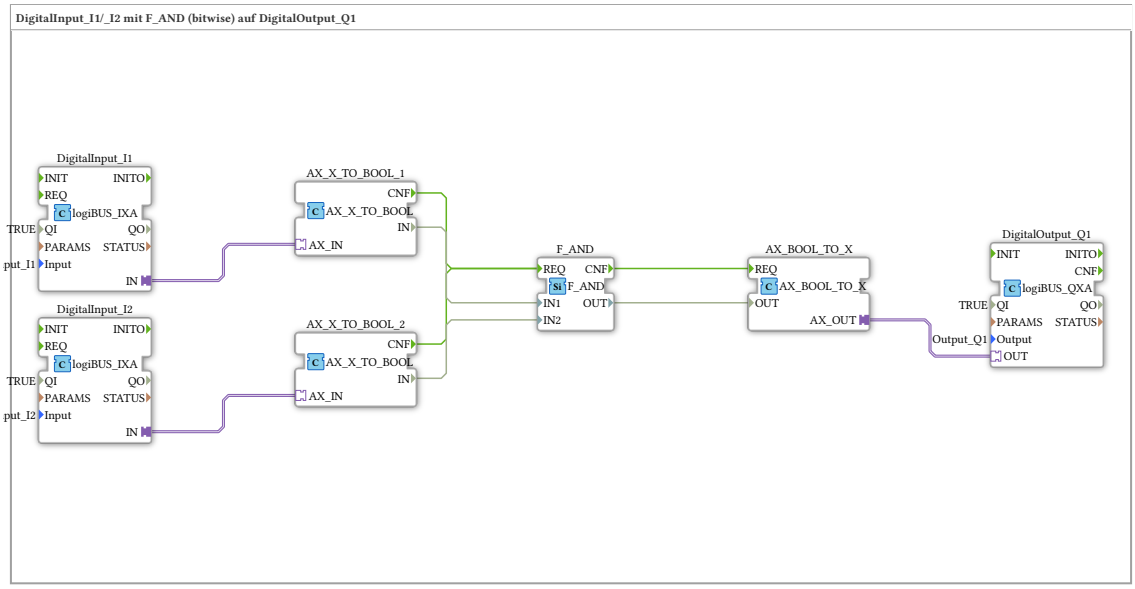
Ziel der Übung

Das Hauptziel ist die Demonstration der Interoperabilität. Während spezialisierte Bausteine wie `AX_AND_2` direkt auf Adaptern arbeiten, benötigen viele Standard-Bibliotheken (wie die bitweisen Operatoren der IEC 61131) elementare Datentypen (BOOL). Diese Übung zeigt den Weg von der Hardware-Abstraktion zur klassischen Logik und zurück.

Beschreibung und Komponenten

Die Subapplikation `Uebung_002a2_AX.SUB` nutzt Konvertierungsbausteine, um zwei Eingangs-Adapter für ein UND-Gatter aufzubereiten.

Funktionsbausteine (FBs)



- **`AX_X_TO_BOOL_1` & `_2`**: Wandeln das Adapter-Signal (`Event + Data`) in ein explizites Ereignis `CNF` und einen booleschen Wert `IN` um.
- **`F_AND`**: Ein klassisches bitweises UND-Gatter aus der IEC 61131 Bibliothek.
- **`AX_BOOL_TO_X`**: Wandelt das Ergebnis der Logik wieder in ein Adapter-Signal um.
- **`DigitalInput_I1` & `I2`**: Eingänge.

- **`DigitalOutput_Q1`**: Ausgang.

Funktionsweise

1. **Erfassung**: Die Adapter-Eingänge liefern bei jeder Änderung ein Signal.
2. **Wandlung**: Die `TO_BOOL` Bausteine extrahieren den Zustand.
3. **Verarbeitung**: Das `F_AND` Gatter prüft: Sind beide Eingänge `TRUE`?
4. **Rückwandlung**: Das Ergebnis wird wieder in die Adapter-Struktur verpackt.
5. **Ausgabe**: Der Ausgang `Q1` schaltet entsprechend.

Diese Methode ist zwar aufwendiger als die Nutzung von `AX_AND_2`, ermöglicht aber den Einsatz jeder beliebigen Logikbibliothek.

Anwendungsbeispiel

Zustandsüberwachung mit Standard-FBs: Wenn Sie komplexe mathematische oder logische Funktionen nutzen möchten, die nur für `BOOL` oder `INT` Datentypen existieren, ist diese Form der Wandlung der Standardweg.

2.1.2.17 Uebung_002a3_AX: DigitalInput_I1/_I2 mit OR auf DigitalOutput_Q1, mit Plug and Socket

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_002a3_AX`. In dieser Übung wird eine logische ODER-Verknüpfung (OR) implementiert, bei der ein digitaler Ausgang aktiviert wird, sobald mindestens einer von zwei digitalen Eingängen den Zustand „Wahr“ (HIGH) einnimmt.

Ziel der Übung

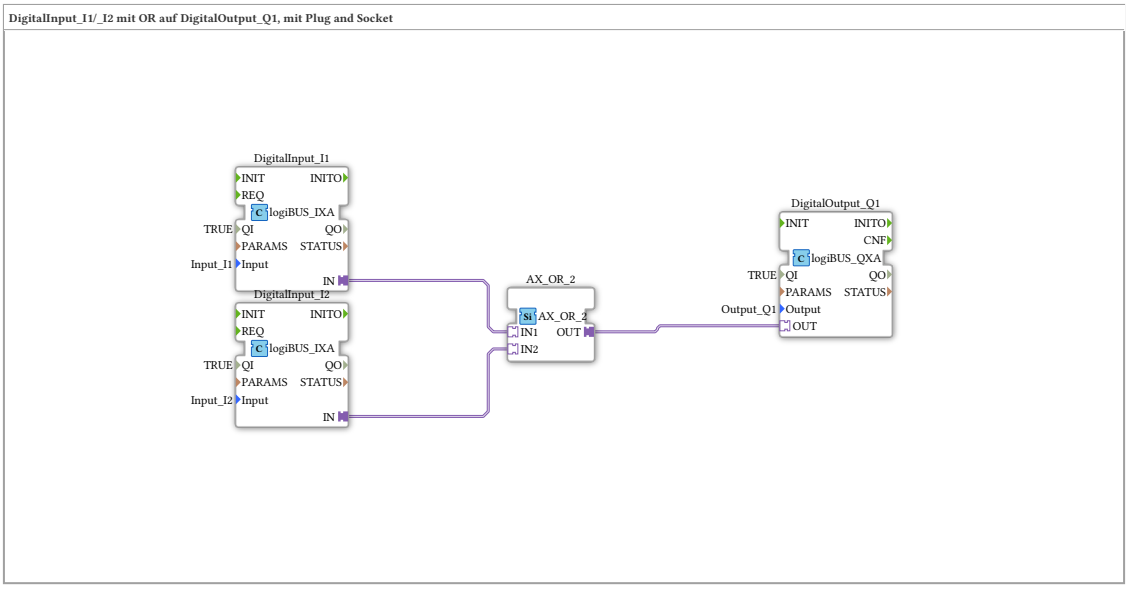
Das Hauptziel dieser Übung ist es, die Funktionsweise einer ODER-Verknüpfung in der Automatisierungstechnik zu demonstrieren. Sie zeigt, wie alternative Bedingungen (Eingänge) genutzt werden können, um dieselbe Aktion (Ausgang) auszulösen. Dies ist eine Standardanforderung für Systeme, die von mehreren Stellen aus bedienbar sein müssen.

Beschreibung und Komponenten

Die Subapplikation `Uebung_002a3_AX.SUB` führt zwei digitale Eingangssignale über einen ODER-Logik-Baustein zusammen.

Funktionsbausteine (FBs)

Folgende Bausteine werden verwendet:



- **`DigitalInput_I1` & `DigitalInput_I2`**: Instanzen des Typs `logiBUS_IXA`. Diese Bausteine erfassen die Zustände der physischen Eingänge `Input_I1` und `Input_I2`.
- **`AX_OR_2`**: Eine Instanz des Typs `AX_OR_2`. Dieser Baustein führt die logische ODER-Operation auf Adapter-Ebene aus. Er besitzt zwei Adapter-Eingänge (`IN1`, `IN2`) und einen Adapter-Ausgang (`OUT`).
- **`DigitalOutput_Q1`**: Eine Instanz des Typs `logiBUS_QXA`. Dieser Baustein setzt den physischen Ausgang `Output_Q1` basierend auf dem Ergebnis der ODER-Verknüpfung.

Adapter-Schnittstelle: `AX.adp`

Durch die Verwendung des Adapter-Typs `AX` werden die Zustandsänderungen (Events) und die booleschen Werte (Daten) gemeinsam durch die Logikbausteine gereicht.

Funktionsweise

Die Logik wird durch die Verschaltung der Adapter-Anschlüsse in der Subapplikation definiert. Der Aufbau in `Uebung_002a3_AX.SUB` ist wie folgt:

```
<AdapterConnections>
    <Connection Source="DigitalInput_I1.IN" Destination="AX_OR_2.IN1"/>
    <Connection Source="DigitalInput_I2.IN" Destination="AX_OR_2.IN2"/>
    <Connection Source="AX_OR_2.OUT" Destination="DigitalOutput_Q1.OUT"/>
</AdapterConnections>
```

Der Prozess folgt dieser Logik:

1. Der Baustein `AX_OR_2` überwacht beide Adapter-Eingänge.
2. Wenn mindestens ein Eingang (`IN1` OR `IN2`) den Datenwert `D1 = TRUE` führt, setzt der Baustein seinen Ausgang `OUT` ebenfalls auf `TRUE` und sendet ein Ereignis.
3. Nur wenn beide Eingänge auf `FALSE` stehen, geht auch der Ausgang auf `FALSE`.
4. Der Baustein `DigitalOutput_Q1` aktualisiert den physischen Ausgang `Q1` bei jeder Änderung am Ausgang des ODER-Bausteins.

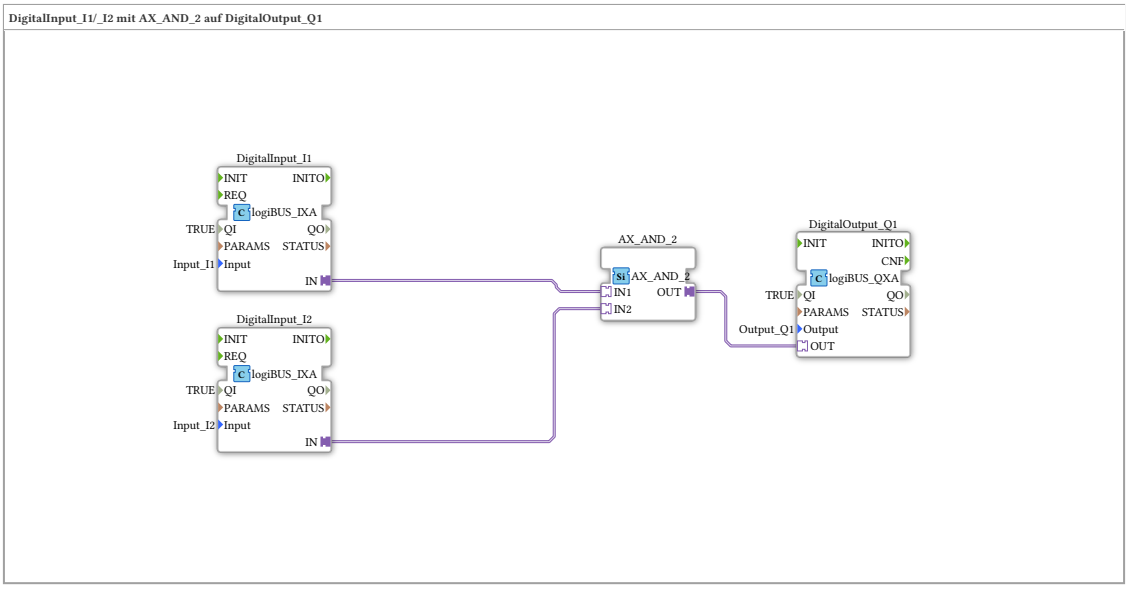
Anwendungsbeispiel

Ein typisches Anwendungsbeispiel ist die **Flurbeleuchtung mit zwei Schaltern**:

In einem langen Flur gibt es an beiden Enden einen Schalter (`I1` und `I2`). Die Lampe (`Q1`) soll leuchten, wenn Schalter 1 betätigt wird ODER wenn Schalter 2 betätigt wird. Diese „Entweder-Oder“-Logik (bzw. „Mindestens-Eins“-Logik) wird durch den `AX_OR_2`-Baustein perfekt abgebildet, sodass man die Beleuchtung von beiden Stellen aus unabhängig voneinander einschalten kann.

2.1.2.18 Uebung_002a4_AX: DigitalInput_I1/_I2 mit AX_AND_2 auf DigitalOutput_Q1

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_002a4_AX (DigitalInput_I1/_I2 mit AX_AND_2 auf DigitalOutput_Q1).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **DigitalInput_I1/_I2 mit AX_AND_2 auf DigitalOutput_Q1**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_002a4_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
- **DigitalInput_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
- **DigitalInput_I2**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I2
- **AX_AND_2**: Instanz des Typs adapter::booleanOperators::AX_AND_2.

Verbindungen und Schnittstellen

Adapterverbindungen:

- DigitalInput_I1.IN -> AX_AND_2.IN1
- DigitalInput_I2.IN -> AX_AND_2.IN2
- AX_AND_2.OUT -> DigitalOutput_Q1.OUT

Programmablauf und Funktionsweise

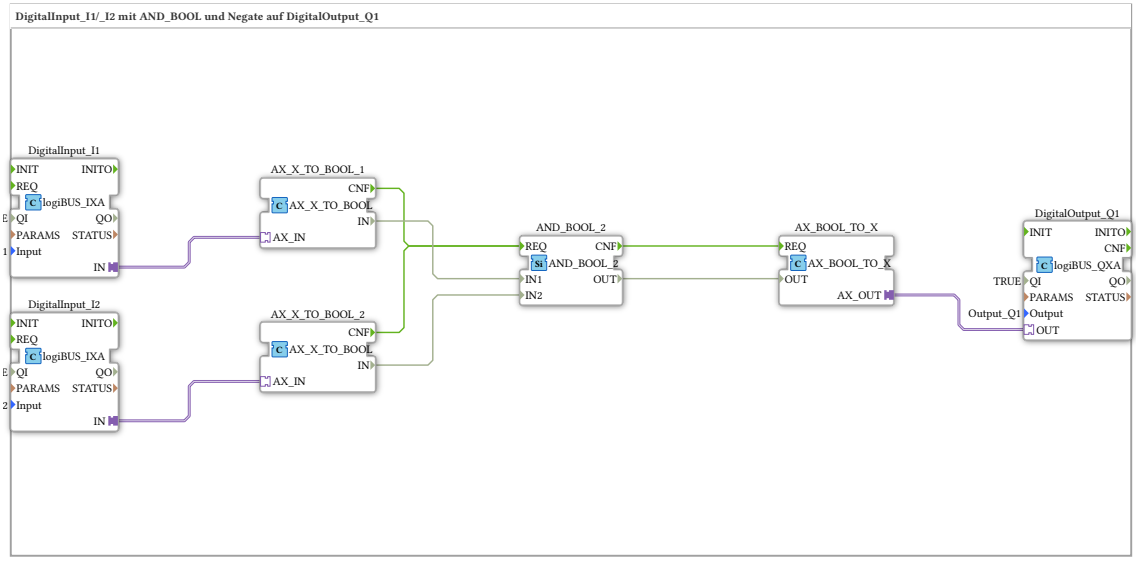
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_002a4_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.19 Uebung_002a4b_AX: DigitalInput_I1/_I2 mit AND_BOOL und Negate auf DigitalOutput_Q1

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_002a4b_AX (DigitalInput_I1/_I2 mit AND_BOOL und Negate auf DigitalOutput_Q1).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **DigitalInput_I1/_I2 mit AND_BOOL und Negate auf DigitalOutput_Q1**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_002a4b_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
- **DigitalInput_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
- **DigitalInput_I2**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I2
- **AX_X_TO_BOOL_1**: Instanz des Typs adapter::conversion::unidirectional::AX_X_TO_BOOL.
- **AX_X_TO_BOOL_2**: Instanz des Typs adapter::conversion::unidirectional::AX_X_TO_BOOL.
- **AND_BOOL_2**: Instanz des Typs iec61131::booleanOperators::AND_BOOL_2.
- **AX_BOOL_TO_X**: Instanz des Typs adapter::conversion::unidirectional::AX_BOOL_TO_X.

Verbindungen und Schnittstellen

Adapterverbindungen:

- DigitalInput_I1.IN -> AX_X_TO_BOOL_1.AX_IN
- DigitalInput_I2.IN -> AX_X_TO_BOOL_2.AX_IN
- AX_BOOL_TO_X.AX_OUT -> DigitalOutput_Q1.OUT

Ereignisverbindungen:

- AX_X_TO_BOOL_1.CNF -> AND_BOOL_2.REQ
- AX_X_TO_BOOL_2.CNF -> AND_BOOL_2.REQ
- AND_BOOL_2.CNF -> AX_BOOL_TO_X.REQ

Datenverbindungen:

- AX_X_TO_BOOL_1.IN -> AND_BOOL_2.IN1
- AX_X_TO_BOOL_2.IN -> AND_BOOL_2.IN2
- AND_BOOL_2.OUT -> AX_BOOL_TO_X.OUT

Hinweise aus dem Modell

„Negate Connection“ mach ein NOT am Eingang; das geht nur bei BOOL.

Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_002a4b_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.20 Uebung_002a5_AX: DigitalInput_I1-3 mit OR auf DigitalOutput_Q1, mit Plug and Socket

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_002a5_AX`. In dieser Übung wird eine logische ODER-Verknüpfung (OR) mit drei Eingängen realisiert. Der digitale Ausgang wird aktiviert, sobald mindestens einer der drei überwachten Eingänge ein Signal führt.

Ziel der Übung

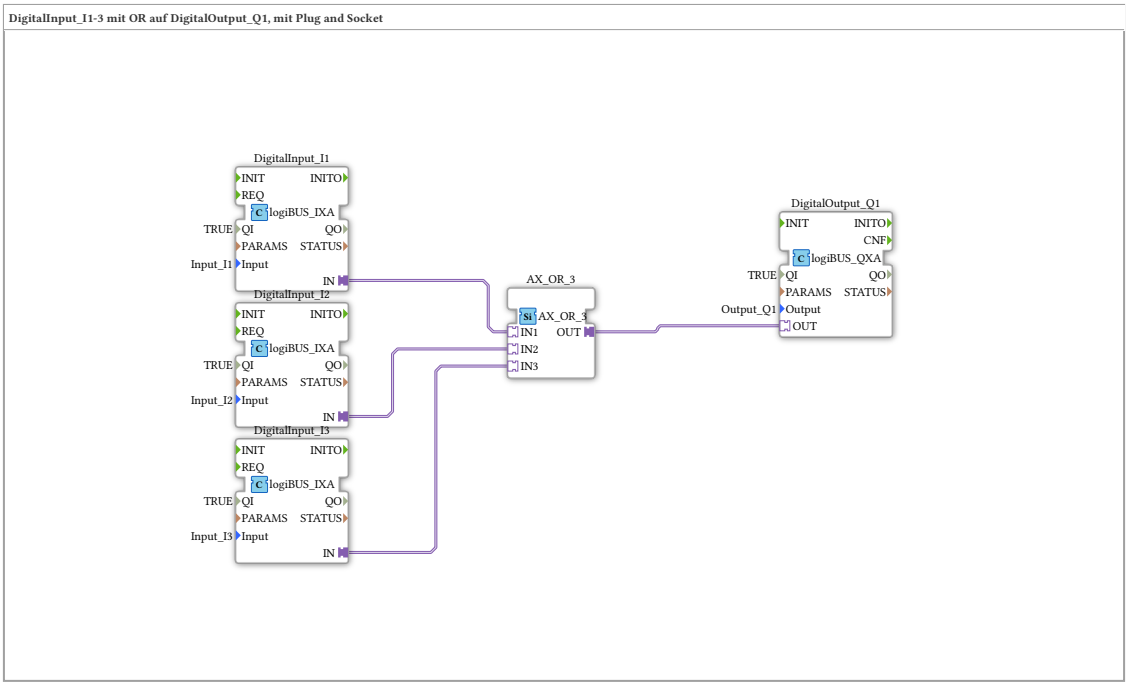
Das Hauptziel dieser Übung ist die Erweiterung der logischen Grundfunktionen auf mehr als zwei Eingangssignale. Sie verdeutlicht die Skalierbarkeit von Logikbausteinen in der IEC 61499 und zeigt, wie mehrere alternative Schaltbedingungen effizient in einer Steuerung zusammengefasst werden können.

Beschreibung und Komponenten

Die Subapplikation `Uebung_002a5_AX.SUB` implementiert eine 3-fach-ODER-Logik unter Verwendung von Adapterverbindungen.

Funktionsbausteine (FBs)

In dieser Konfiguration werden folgende Bausteine eingesetzt:



- **`DigitalInput_I1, I2, I3`**: Drei Instanzen des Typs `logiBUS_IXA`. Diese erfassen die Zustände der Hardware-Eingänge `Input_I1` bis `Input_I3`.
- **`AX_OR_3`**: Eine Instanz des Typs `AX_OR_3`. Dieser Baustein führt eine ODER-Verknüpfung für drei Adapter-Eingänge (`IN1`, `IN2`, `IN3`) aus und stellt das Ergebnis am Adapter-Ausgang `OUT` bereit.
- **`DigitalOutput_Q1`**: Eine Instanz des Typs `logiBUS_QXA`. Dieser Baustein steuert den Hardware-Ausgang `Output_Q1`.

Adapter-Schnittstelle: `AX.adp`

Wie bei den vorangegangenen Übungen wird der Adapter-Typ `AX` für die nahtlose Übertragung von Ereignissen und Daten verwendet.

Funktionsweise

Die Logik wird durch die Verschaltung der drei Eingänge mit dem Logik-Baustein in der Subapplikation realisiert. Der Aufbau in `Uebung_002a5_AX.SUB` ist wie folgt definiert:

```
<AdapterConnections>
    <Connection Source="DigitalInput_I1.IN" Destination="AX_OR_3.IN1"/>
    <Connection Source="DigitalInput_I2.IN" Destination="AX_OR_3.IN2"/>
    <Connection Source="DigitalInput_I3.IN" Destination="AX_OR_3.IN3"/>
    <Connection Source="AX_OR_3.OUT" Destination="DigitalOutput_Q1.OUT"/>
</AdapterConnections>
```

Der funktionale Ablauf:

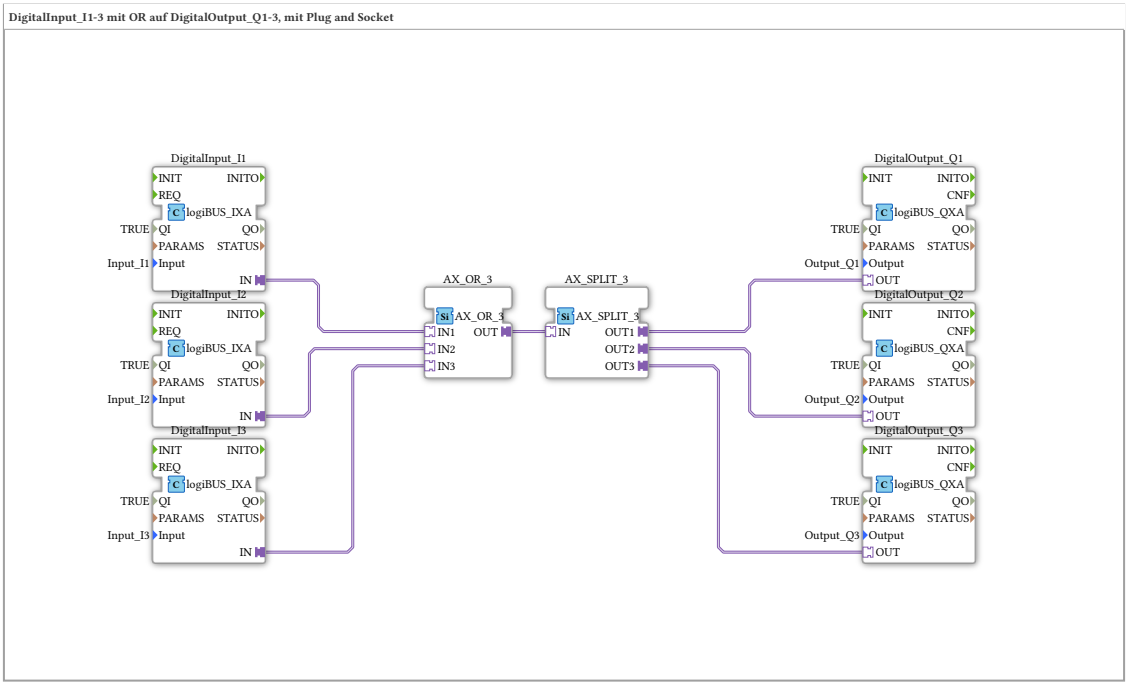
1. Der Baustein `AX_OR_3` überwacht kontinuierlich alle drei Adapter-Eingänge auf Zustandsänderungen.
2. Wenn mindestens ein Eingang den Zustand `TRUE` einnimmt, schaltet der Ausgang `OUT` auf `TRUE`.
3. Nur wenn alle drei Eingänge (`I1` UND `I2` UND `I3`) den Zustand `FALSE` haben, wird auch der Ausgang deaktiviert.
4. Der Ausgangsbaustein `DigitalOutput_Q1` folgt dem logischen Ergebnis des ODER-Bausteins in Echtzeit.

Anwendungsbeispiel

Ein typisches Anwendungsbeispiel ist eine **Sammelstörmeldung**:

Drei verschiedene Sensoren an einer Maschine (z.B. Übertemperatur `I1`, Ölmangel `I2` und Not-Halt `I3`) sollen eine gemeinsame Warnlampe (`Q1`) oder eine Hupe aktivieren. Sobald auch nur einer der Sensoren eine Störung meldet, wird der Bediener über den gemeinsamen Ausgang gewarnt. Dies reduziert den Verdrahtungsaufwand und bündelt wichtige Statusinformationen.

2.1.2.21 Uebung_002a5b_AX: DigitalInput_I1-3 mit OR auf DigitalOutput_Q1-3, mit Plug and Socket



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung demonstriert die grundlegende Verknüpfung von mehreren digitalen Eingängen mit mehreren digitalen Ausgängen. Dabei wird eine logische OR-Operation eingesetzt, um die Zustände der Eingänge zu verknüpfen. Das Ergebnis dieser Verknüpfung wird anschließend über einen Signalverteiler auf verschiedene digitale Ausgänge verteilt. Die Implementierung nutzt dabei das Konzept von Adapter-Funktionsbausteinen, um die boolesche Logik und die Signalverteilung zu realisieren.

Verwendete Funktionsbausteine (FBs)

Die Übung `Uebung_002a5b_AX` verwendet eine Kombination aus spezifischen I/O-Bausteinen und generischen Logik- sowie Signalverteilungs-Bausteinen.

Sub-Bausteine: logiBUS_IXA

- **Typ**: `logiBUS::io::DI::logiBUS_IXA` (repräsentiert durch Instanzen wie `DigitalInput_I1`, `DigitalInput_I2`, `DigitalInput_I3`)
- **Verwendete interne FBs**: Keine internen FBs in der bereitgestellten Definition sichtbar.
  - **Bausteinname**: DigitalInput_I1 (Beispielhafte Instanz)
    - Parameter: QI = TRUE
    - Parameter: Input = Input_I1
    - Ereignisausgang/-eingang: Wird intern für die Verarbeitung des Eingangsstatus genutzt; sendet typischerweise ein Datenereignis bei Wertänderung.
    - Datenausgang/-eingang: IN (Datenausgang, der den logischen Zustand des digitalen Eingangs liefert)
- **Funktionsweise**: Dieser Funktionsbaustein dient zum Einlesen des logischen Zustands eines spezifischen digitalen Eingangs. Er überwacht den zugewiesenen physikalischen Eingang (z.B. `Input_I1`) und stellt dessen aktuellen Status als booleschen Wert an seinem Datenausgang `IN` bereit.

Sub-Bausteine: logiBUS_QXA

- **Typ**: `logiBUS::io::DQ::logiBUS_QXA` (repräsentiert durch Instanzen wie `DigitalOutput_Q1`, `DigitalOutput_Q2`, `DigitalOutput_Q3`)
- **Verwendete interne FBs**: Keine internen FBs in der bereitgestellten Definition sichtbar.
  - **Bausteinname**: DigitalOutput_Q1 (Beispielhafte Instanz)

    - Parameter: QI = TRUE
    - Parameter: Output = Output_Q1
    - Ereignisausgang/-eingang: Wird intern für die Verarbeitung des Ausgangsstatus genutzt; empfängt typischerweise ein Datenereignis zur Aktualisierung des Ausgangs.
    - Datenausgang/-eingang: OUT (Dateneingang, der den logischen Zustand zum Setzen des digitalen Ausgangs empfängt)
- **Funktionsweise**: Dieser Funktionsbaustein dient zur Ansteuerung eines spezifischen digitalen Ausgangs. Er setzt den Zustand des zugewiesenen physikalischen Ausgangs (z.B. `Output_Q1`) basierend auf dem booleschen Wert, der an seinem Dateneingang `OUT` anliegt.

Sub-Bausteine: AX_OR_3

- **Typ**: `adapter::booleanOperators::AX_OR_3` (repräsentiert durch die Instanz `AX_OR_3`)
- **Verwendete interne FBs**: Keine internen FBs in der bereitgestellten Definition sichtbar.
  - **Bausteinname**: AX_OR_3
    - Parameter: Keine spezifischen Parameter für diese Instanz in der Definition vorhanden.
    - Ereignisausgang/-eingang: Überträgt Ereignisse synchron mit den Daten (Plug and Socket Adapter).
    - Datenausgang/-eingang: IN1, IN2, IN3 (Dateneingänge), OUT (Datenausgang)
- **Funktionsweise**: Dieser Baustein implementiert eine logische OR-Verknüpfung mit drei Eingängen. Der Datenausgang `OUT` wird `TRUE`, wenn mindestens einer der drei Dateneingänge (`IN1`, `IN2`, `IN3`) den Wert `TRUE` hat. Andernfalls ist der Ausgang `FALSE`.

Sub-Bausteine: AX_SPLIT_3

- **Typ**: `adapter::events::unidirectional::AX_SPLIT_3` (repräsentiert durch die Instanz `AX_SPLIT_3`)
- **Verwendete interne FBs**: Keine internen FBs in der bereitgestellten Definition sichtbar.
  - **Bausteinname**: AX_SPLIT_3
    - Parameter: Keine spezifischen Parameter für diese Instanz in der Definition vorhanden.
    - Ereignisausgang/-eingang: Überträgt Ereignisse synchron mit den Daten (Plug and Socket Adapter).
    - Datenausgang/-eingang: IN (Dateneingang), OUT1, OUT2, OUT3 (Datenausgänge)
- **Funktionsweise**: Dieser Baustein dient als Signalverteiler. Er nimmt ein einzelnes Eingangssignal am Dateneingang `IN` entgegen und leitet es identisch und gleichzeitig an drei separate Datenausgänge (`OUT1`, `OUT2`, `OUT3`) weiter.

Programmablauf und Verbindungen

Die Übung `Uebung_002a5b_AX` realisiert eine Steuerungslogik, bei der die Zustände von drei digitalen Eingängen über eine OR-Verknüpfung ausgewertet und das Ergebnis auf drei digitale Ausgänge verteilt wird.

1. **Erfassung der Eingänge**: Die Funktionsbausteine `DigitalInput_I1`, `DigitalInput_I2` und `DigitalInput_I3` lesen kontinuierlich die Zustände der physikalischen Eingänge `Input_I1`, `Input_I2` und `Input_I3` ein. Ihre jeweiligen Datenausgänge (`DigitalInput_I1.IN`, `DigitalInput_I2.IN`, `DigitalInput_I3.IN`) stellen diese Zustände bereit.
2. **Logische OR-Verknüpfung**: Die Datenausgänge der drei Eingangsbausteine werden direkt mit den Dateneingängen des OR-Bausteins `AX_OR_3` verbunden:
   - `DigitalInput_I1.IN` ist mit `AX_OR_3.IN1` verbunden.
   - `DigitalInput_I2.IN` ist mit `AX_OR_3.IN2` verbunden.
   - `DigitalInput_I3.IN` ist mit `AX_OR_3.IN3` verbunden. Der `AX_OR_3` Baustein verknüpft diese drei booleschen Werte logisch. Das Ergebnis (`TRUE`, wenn I1 ODER I2 ODER I3 `TRUE` ist) wird an seinem Datenausgang `AX_OR_3.OUT` zur Verfügung gestellt.
3. **Signalverteilung**: Der Datenausgang des OR-Bausteins (`AX_OR_3.OUT`) wird an den Dateneingang des Signalverteilers `AX_SPLIT_3` (`AX_SPLIT_3.IN`) angeschlossen. Der `AX_SPLIT_3` Baustein dupliziert dieses einzelne Steuersignal und leitet es an seine drei Datenausgänge (`AX_SPLIT_3.OUT1`, `AX_SPLIT_3.OUT2`, `AX_SPLIT_3.OUT3`) weiter.
4. **Ansteuerung der Ausgänge**: Die Ausgänge des Signalverteilers sind jeweils mit den Eingängen der digitalen Ausgangsbausteine verbunden:
   - `AX_SPLIT_3.OUT1` ist mit `DigitalOutput_Q1.OUT` verbunden.
   - `AX_SPLIT_3.OUT2` ist mit `DigitalOutput_Q2.OUT` verbunden.

- `AX_SPLIT_3.OUT3` ist mit `DigitalOutput_Q3.OUT` verbunden. Dies bedeutet, dass alle drei digitalen Ausgänge `Output_Q1`, `Output_Q2` und `Output_Q3` den gleichen Zustand annehmen, der dem Ergebnis der OR-Verknüpfung der drei Eingänge entspricht.

Lernziele:

- Verständnis und Anwendung von digitalen Eingangs- und Ausgangsbausteinen.
- Implementierung grundlegender logischer Operationen (OR) in 4diac-IDE.
- Nutzung von Signalverteilern (Splittern) zur effizienten Ansteuerung mehrerer Komponenten von einem einzigen Steuersignal.
- Kennenlernen des Konzepts von Adapter-Bausteinen für flexible Verbindungen.

Schwierigkeitsgrad: Mittel. Grundkenntnisse in digitaler Logik und der Bedienung der 4diac-IDE sind von Vorteil.

Benötigte Vorkenntnisse: Vertrautheit mit den Grundlagen der Automatisierungstechnik und der Entwicklung von Anwendungen in der 4diac-IDE.

Start der Übung: Laden Sie die Applikation `Uebung_002a5b_AX` auf eine 4diac-kompatible Steuerung (SPS oder Laufzeitumgebung). Beobachten Sie das Verhalten der Ausgänge Q1, Q2 und Q3, wenn Sie die digitalen Eingänge I1, I2 oder I3 manuell schalten.

Zusammenfassung

Die Übung `Uebung_002a5b_AX` bietet eine praktische Einführung in die Verknüpfung von digitalen I/O-Signalen. Sie demonstriert, wie mithilfe einer logischen OR-Operation mehrere Eingänge zu einem einzigen Steuersignal zusammengefasst werden können. Dieses Signal wird anschließend gesplittet, um eine synchronisierte Ansteuerung von mehreren Ausgängen zu ermöglichen. Das Kernprinzip ist, dass alle drei Ausgänge (Q1, Q2, Q3) aktiv werden, sobald mindestens einer der drei Eingänge (I1, I2, I3) aktiv ist. Diese Art der Gruppensteuerung ist eine grundlegende Funktion in vielen Automatisierungsanwendungen.

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

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

2.1.2.22 Uebung_002a6_AX: DigitalInput_I1-3 mit AND auf DigitalOutput_Q1, mit Plug and Socket



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_002a6_AX`. In dieser Übung wird eine logische UND-Verknüpfung (AND) mit drei Eingängen implementiert. Der digitale Ausgang wird nur dann aktiviert, wenn alle drei überwachten Eingänge gleichzeitig den Zustand „Wahr“ (HIGH) führen.

Ziel der Übung

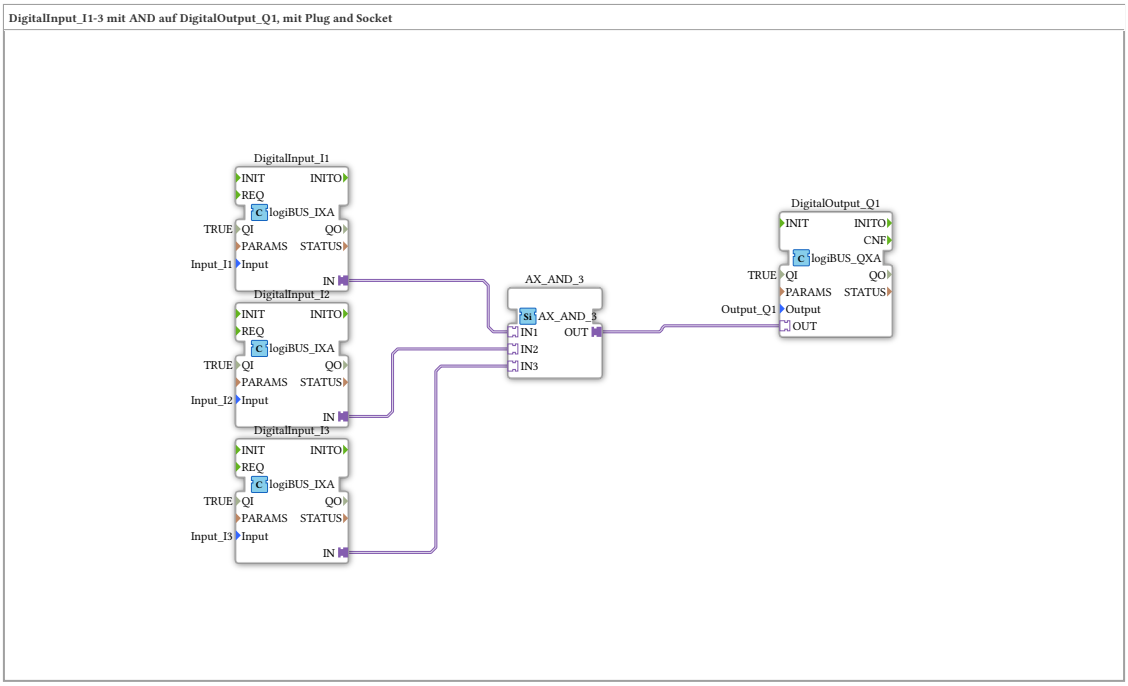
Das Hauptziel dieser Übung ist die Umsetzung einer komplexeren Bedingungslogik. Es wird demonstriert, wie mehrere Sicherheits- oder Betriebsparameter kombiniert werden können, um eine Freigabe für einen Aktor zu erteilen. Dies ist eine typische Anforderung in der industriellen Steuerungstechnik, um sicherzustellen, dass mehrere Voraussetzungen erfüllt sind, bevor eine Aktion ausgeführt wird.

Beschreibung und Komponenten

Die Subapplikation `Uebung_002a6_AX.SUB` nutzt einen 3-fach-UND-Baustein, um drei digitale Eingänge mit einem Ausgang zu verknüpfen.

Funktionsbausteine (FBs)

Folgende Bausteine werden verwendet:



- **`DigitalInput_I1, I2, I3`**: Drei Instanzen des Typs `logiBUS_IXA`. Diese erfassen die Zustände der physischen Eingänge `Input_I1` bis `Input_I3`.
- **`AX_AND_3`**: Eine Instanz des Typs `AX_AND_3`. Dieser Baustein führt die logische UND-Operation für drei Adapter-Eingänge (`IN1`, `IN2`, `IN3`) aus und stellt das Ergebnis am Adapter-Ausgang `OUT` bereit.
- **`DigitalOutput_Q1`**: Eine Instanz des Typs `logiBUS_QXA`. Dieser Baustein steuert den Hardware-Ausgang `Output_Q1`.

Adapter-Schnittstelle: `AX.adp`

Wie in den vorherigen Übungen wird der Adapter-Typ `AX` verwendet, um Ereignisse und Datenwerte gekapselt durch die Logik zu leiten.

Funktionsweise

Die Logik wird durch die Verschaltung der drei Eingangsbausteine mit dem UND-Logik-Baustein in der Subapplikation realisiert. Der Aufbau in `Uebung_002a6_AX.SUB` ist wie folgt definiert:

```
<AdapterConnections>
    <Connection Source="DigitalInput_I1.IN" Destination="AX_AND_3.IN1"/>
    <Connection Source="DigitalInput_I2.IN" Destination="AX_AND_3.IN2"/>
    <Connection Source="DigitalInput_I3.IN" Destination="AX_AND_3.IN3"/>
    <Connection Source="AX_AND_3.OUT" Destination="DigitalOutput_Q1.OUT"/>
</AdapterConnections>
```

Der funktionale Ablauf:

1. Der Baustein `AX_AND_3` überwacht alle drei Adapter-Eingänge auf Zustandsänderungen.
2. Nur wenn alle drei Eingänge (`I1` AND `I2` AND `I3`) gleichzeitig den Datenwert `D1 = TRUE` führen, wird der Ausgang `OUT` ebenfalls auf `TRUE` gesetzt.
3. Sobald auch nur einer der drei Eingänge auf `FALSE` geht, wird der Ausgang sofort deaktiviert.
4. Der Baustein `DigitalOutput_Q1` schaltet den physischen Ausgang `Q1` entsprechend dem logischen Ergebnis.

Anwendungsbeispiel

Ein typisches Anwendungsbeispiel ist eine **Maschinenfreigabe mit mehreren Sicherheitsbedingungen**:

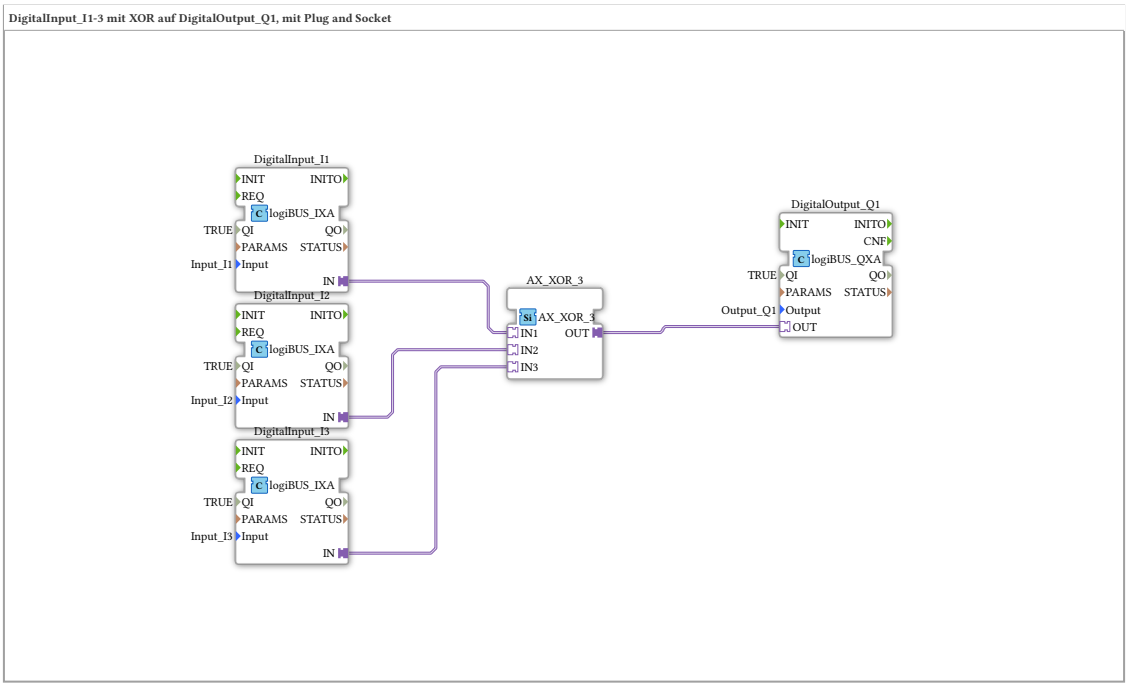
Damit eine Maschine (`Q1`) starten darf, müssen drei Bedingungen erfüllt sein: Die Schutztür muss geschlossen sein (`I1`), der Not-Halt muss entriegelt sein (`I2`) und der Bediener muss den Starttaster drücken (`I3`). Erst wenn alle drei Signale gleichzeitig „Wahr“ sind, gibt die Steuerung den Betrieb frei. Dies gewährleistet ein Höchstmaß an Sicherheit für Mensch und Maschine.

2.1.2.23 Uebung_002a7_AX: DigitalInput_I1-3 mit XOR auf DigitalOutput_Q1, mit Plug and Socket



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_002a7_AX`. In dieser Übung wird eine exklusive ODER-Verknüpfung (XOR) mit drei Eingängen realisiert. Der Ausgang wird aktiviert, wenn eine ungerade Anzahl von Eingängen aktiv ist.

Ziel der Übung



Das Hauptziel dieser Übung ist die Demonstration der XOR-Logik bei mehr als zwei Eingängen. Im Gegensatz zur normalen ODER-Verknüpfung, bei der der Ausgang bei *mindestens* einem aktiven Eingang einschaltet, reagiert die XOR-Logik auf die Parität der Eingangssignale. Dies wird oft für Wechselschaltungen oder Paritätsprüfungen verwendet.

Beschreibung und Komponenten

Die Subapplikation `Uebung_002a7_AX.SUB` nutzt einen 3-fach-XOR-Baustein, um drei digitale Eingänge mit einem Ausgang zu verknüpfen.

Funktionsbausteine (FBs)

Folgende Bausteine werden eingesetzt:

- **`DigitalInput_I1`, `I2`, `I3`**: Drei Instanzen des Typs `logiBUS_IXA`. Diese erfassen die Hardware-Eingänge `Input_I1` bis `Input_I3`.
- **`AX_XOR_3`**: Eine Instanz des Typs `AX_XOR_3`. Dieser Baustein führt die exklusive ODER-Operation auf drei Adapter-Eingängen (`IN1`, `IN2`, `IN3`) aus und stellt das Ergebnis am Adapter-Ausgang `OUT` bereit.
- **`DigitalOutput_Q1`**: Eine Instanz des Typs `logiBUS_QXA`. Dieser Baustein steuert den Hardware-Ausgang `Output_Q1`.

Adapter-Schnittstelle: `AX.adp`

Der Adapter-Typ `AX` bündelt auch hier Ereignis und Datenwert für eine effiziente Logikverarbeitung.

Funktionsweise

Die Logik wird durch die Verschaltung der Eingangsbausteine mit dem XOR-Logik-Baustein in der Subapplikation definiert. Der Aufbau in `Uebung_002a7_AX.SUB` sieht wie folgt aus:

```
<AdapterConnections>
    <Connection Source="DigitalInput_I1.IN" Destination="AX_XOR_3.IN1"/>
    <Connection Source="DigitalInput_I2.IN" Destination="AX_XOR_3.IN2"/>
    <Connection Source="DigitalInput_I3.IN" Destination="AX_XOR_3.IN3"/>
    <Connection Source="AX_XOR_3.OUT" Destination="DigitalOutput_Q1.OUT"/>
</AdapterConnections>
```

Die XOR-Logik mit drei Eingängen verhält sich wie folgt:

- Der Ausgang ist **TRUE**, wenn genau **ein** Eingang aktiv ist.
- Der Ausgang ist **TRUE**, wenn alle **drei** Eingänge aktiv sind.
- Der Ausgang ist **FALSE**, wenn kein Eingang oder genau zwei Eingänge aktiv sind.

Dies entspricht der mathematischen Definition der XOR-Verknüpfung als ungerade Parität.

Anwendungsbeispiel

Ein klassisches Anwendungsbeispiel ist eine **Kreuzschaltung mit drei Schaltern**:

In einem Raum mit drei Türen gibt es an jeder Tür einen Schalter (`I1`, `I2`, `I3`). Das Licht (`Q1`) soll von jeder Tür aus ein- und ausgeschaltet werden können, unabhängig vom Zustand der anderen Schalter. Jede Betätigung eines beliebigen Schalters ändert den Zustand des Lichts (von An zu Aus oder umgekehrt). Dies wird durch die Paritätslogik der XOR-Verknüpfung perfekt realisiert.

2.1.2.24 Uebung_002a_AX: DigitalInput_I1/_I2 mit AND auf DigitalOutput_Q1, mit Plug and Socket



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_002a_AX`. In dieser Übung wird eine klassische UND-Verknüpfung (AND) realisiert, bei der ein digitaler Ausgang nur dann aktiviert wird, wenn zwei digitale Eingänge gleichzeitig den Zustand „Wahr“ (HIGH) führen.

Ziel der Übung

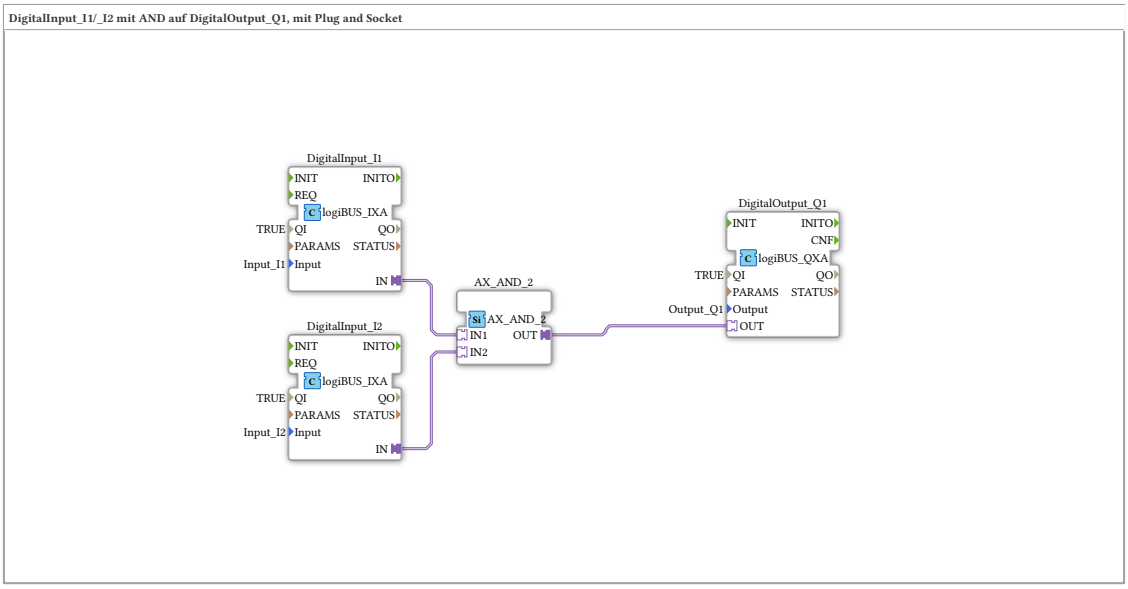
Das Hauptziel dieser Übung ist die Implementierung einer grundlegenden logischen Entscheidungsstruktur. Es wird gezeigt, wie Signale von mehreren Sensoren (Eingängen) kombiniert werden können, um eine Aktion an einem Aktor (Ausgang) auszulösen. Dies ist ein fundamentaler Baustein jeder Steuerungsprogrammierung.

Beschreibung und Komponenten

Die Subapplikation `Uebung_002a_AX.SUB` verknüpft zwei digitale Eingänge über einen Logik-Baustein mit einem digitalen Ausgang.

Funktionsbausteine (FBs)

Folgende Bausteine werden verwendet:



- **`DigitalInput_I1` & `DigitalInput_I2`**: Instanzen des Typs `logiBUS_IXA`. Diese repräsentieren die beiden Hardware-Eingänge, die überwacht werden.
- **`AX_AND_2`**: Eine Instanz des Typs `AX_AND_2`. Dieser Baustein führt die logische UND-Operation direkt auf den Adapter-Schnittstellen aus. Er besitzt zwei Adapter-Eingänge (`IN1`, `IN2`) und einen Adapter-Ausgang (`OUT`).
- **`DigitalOutput_Q1`**: Eine Instanz des Typs `logiBUS_QXA`. Dieser Baustein steuert den Hardware-Ausgang `Output_Q1` basierend auf dem Ergebnis der Logik.

Adapter-Schnittstelle: `AX.adp`

Die gesamte Signalverarbeitung erfolgt über den Adapter-Typ `AX`, wodurch Ereignisse und Datenwerte effizient durch das Netzwerk geleitet werden.

Funktionsweise

Die Logik wird durch die Verschaltung der Adapter-Anschlüsse in der Subapplikation festgelegt. Der Aufbau in `Uebung_002a_AX.SUB` ist wie folgt definiert:

```
<AdapterConnections>
    <Connection Source="DigitalInput_I1.IN" Destination="AX_AND_2.IN1"/>
    <Connection Source="DigitalInput_I2.IN" Destination="AX_AND_2.IN2"/>
    <Connection Source="AX_AND_2.OUT" Destination="DigitalOutput_Q1.OUT"/>
</AdapterConnections>
```

Der Prozess folgt dieser Logik:

1. Der Baustein `AX_AND_2` überwacht beide Adapter-Eingänge.
2. Nur wenn beide Eingänge (`IN1` AND `IN2`) den Datenwert `D1 = TRUE` führen, setzt der Baustein seinen Ausgang `OUT` ebenfalls auf `TRUE` und sendet ein Ereignis.
3. Sobald einer der Eingänge auf `FALSE` geht, wird auch der Ausgang sofort auf `FALSE` gesetzt.
4. Der Baustein `DigitalOutput_Q1` reagiert unmittelbar auf die Zustandsänderungen am Ausgang des Logik-Bausteins.

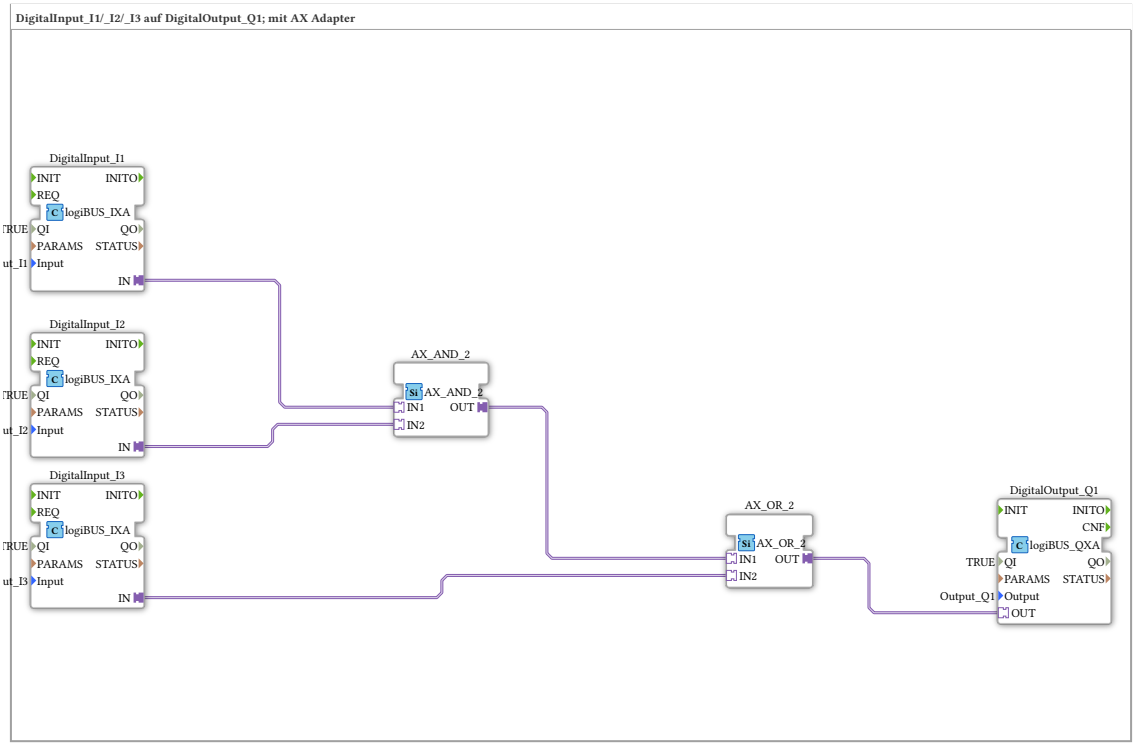
Anwendungsbeispiel

Ein klassisches Anwendungsbeispiel ist die **Zweihandbedienung zur Sicherheit**:

Um eine gefährliche Maschine (z.B. eine Presse) zu starten, muss der Bediener zwei räumlich getrennte Taster (`I1` und `I2`) gleichzeitig drücken. Dies stellt sicher, dass sich beide Hände des Bedieners außerhalb des Gefahrenbereichs befinden. Nur wenn beide Taster aktiv sind, gibt der `AX_AND_2`-Baustein das Signal an den Ausgang `Q1` (den Motor der Presse) frei.

2.1.2.25 Uebung_002b2_AX: DigitalInput_I1/_I2/_I3 auf DigitalOutput_Q1; mit AX Adapter

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_002b2_AX (DigitalInput_I1/_I2/_I3 auf DigitalOutput_Q1; mit AX Adapter).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **DigitalInput_I1/_I2/_I3 auf DigitalOutput_Q1; mit AX Adapter**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_002b2_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
- **DigitalInput_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
- **DigitalInput_I2**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I2
- **AX_AND_2**: Instanz des Typs adapter::booleanOperators::AX_AND_2.
- **DigitalInput_I3**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I3
- **AX_OR_2**: Instanz des Typs adapter::booleanOperators::AX_OR_2.

Verbindungen und Schnittstellen

Adapterverbindungen:

- DigitalInput_I1.IN -> AX_AND_2.IN1
- DigitalInput_I2.IN -> AX_AND_2.IN2

- DigitalInput_I3.IN -> AX_OR_2.IN2
- AX_AND_2.OUT -> AX_OR_2.IN1
- AX_OR_2.OUT -> DigitalOutput_Q1.OUT

Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_002b2_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.26 Uebung_002b3_AX: DigitalInput_I1/_I2/_I3 auf DigitalOutput_Q1



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_002b3_AX`. In dieser Übung wird eine kombinatorische Logikschaltung implementiert, die zwei Grundoperationen (UND und ODER) miteinander verknüpft, um eine komplexere Schaltbedingung zu erfüllen.

Ziel der Übung

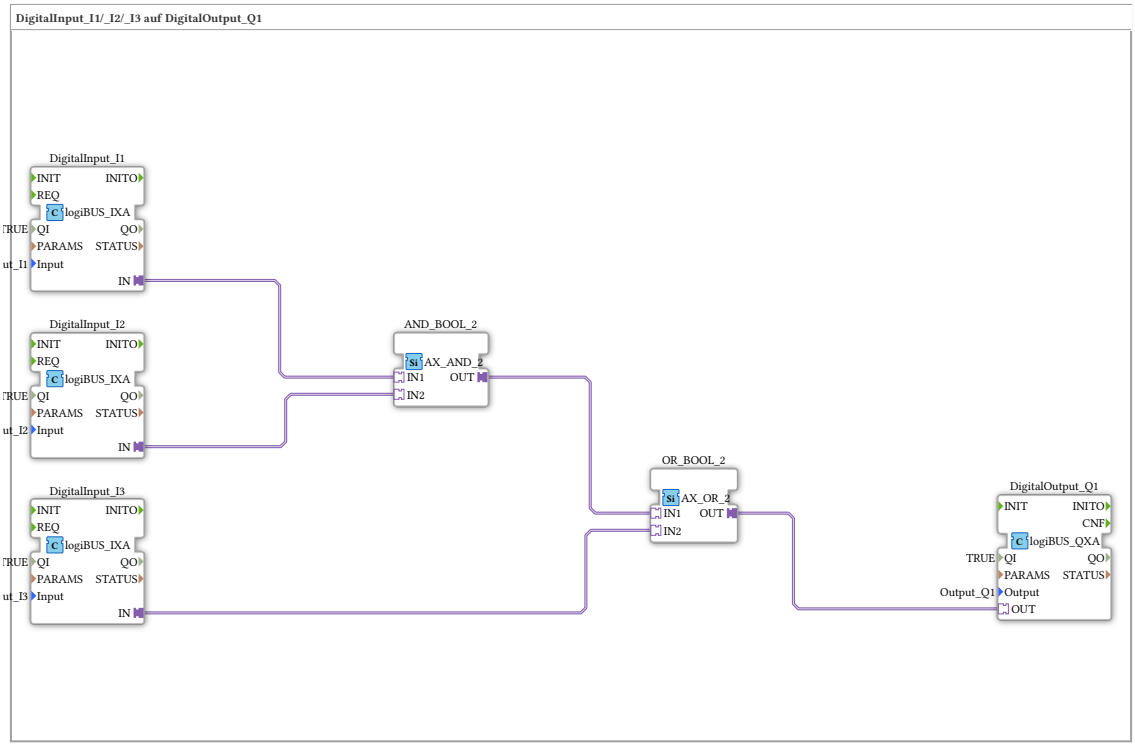
Das Hauptziel dieser Übung ist die hierarchische Verknüpfung von Logikbausteinen. Es wird gezeigt, wie Teilergebnisse einer logischen Operation (hier ein AND) als Eingangsgröße für eine weitere Operation (hier ein OR) dienen können. Dies ermöglicht die Abbildung beliebig komplexer logischer Ausdrücke in der Steuerungstechnik.

Beschreibung und Komponenten

Die Subapplikation `Uebung_002b3_AX.SUB` realisiert die logische Funktion `Q1 = (I1 AND I2) OR I3` unter Verwendung von Adapter-Logikbausteinen.

Funktionsbausteine (FBs)

In der Subapplikation werden folgende Komponenten instanziiert:



- **`DigitalInput_I1, I2, I3`**: Instanzen des Typs `logiBUS_IXA`. Sie liefern die Eingangssignale für die Logikkette.
- **`AND_2_BOOL`**: Eine Instanz des Typs `AX_AND_2`. Verknüpft die Eingänge `I1` und `I2`.
- **`OR_2_BOOL`**: Eine Instanz des Typs `AX_OR_2`. Verknüpft das Ergebnis des UND-Bausteins mit dem dritten Eingang `I3`.
- **`DigitalOutput_Q1`**: Eine Instanz des Typs `logiBUS_QXA`. Gibt das Endergebnis der kombinatorischen Logik an den Hardware-Ausgang aus.

Adapter-Schnittstelle: `AX.adp`

Durch die konsequente Verwendung von Adapter-Bausteinen kann auf explizite Event-Daten-Konverter (wie `F_MOVE`) verzichtet werden, da die `AX`-Bausteine beides intern handhaben.

Funktionsweise

Die hierarchische Struktur der Logik wird durch die Verschaltung der Adapter-Verbindungen in der Subapplikation `Uebung_002b3_AX.SUB` deutlich:

```
<AdapterConnections>
    <Connection Source="DigitalInput_I1.IN" Destination="AND_2_BOOL.IN1"/>
    <Connection Source="DigitalInput_I2.IN" Destination="AND_2_BOOL.IN2"/>
    <Connection Source="AND_2_BOOL.OUT" Destination="OR_2_BOOL.IN1"/>
    <Connection Source="DigitalInput_I3.IN" Destination="OR_2_BOOL.IN2"/>
    <Connection Source="OR_2_BOOL.OUT" Destination="DigitalOutput_Q1.OUT"/>
</AdapterConnections>
```

Der funktionale Ablauf:

1. Das System berechnet zuerst das Teilergebnis der UND-Verknüpfung von `I1` und `I2`.
2. Dieses Teilergebnis wird an den ersten Eingang des ODER-Bausteins gereicht.
3. Der ODER-Baustein vergleicht dieses Teilergebnis mit dem direkten Signal von `I3`.
4. Der Ausgang `Q1` wird aktiviert, wenn entweder beide ersten Eingänge (`I1` AND `I2`) aktiv sind ODER wenn der dritte Eingang (`I3`) aktiv ist.

Anwendungsbeispiel

Ein typisches Anwendungsbeispiel ist eine **Anlagenfreigabe mit Überbrückung**:

Ein Motor (`Q1`) soll normalerweise nur laufen, wenn zwei Sensoren (`I1` und `I2`) gleichzeitig grünes Licht geben (z.B. Öldruck erreicht UND Temperatur ok). Für Wartungszwecke oder im Notbetrieb soll

der Motor jedoch auch dann gestartet werden können, wenn ein spezieller Schlüsselschalter (`I3`) betätigt wird, der die normale Logik überbrückt (Bypass). Diese Anforderung wird durch die `(I1 AND I2) OR I3` Logik exakt erfüllt.

2.1.2.27 Uebung_003_AX: DigitalInput_I1/_I2 auf DigitalOutput_Q1/_I2 - flach mit Adapter



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_003_AX`. In dieser Übung werden zwei voneinander unabhängige Signalpfade realisiert, bei denen jeweils ein digitaler Eingang direkt einen zugeordneten digitalen Ausgang steuert.

Ziel der Übung

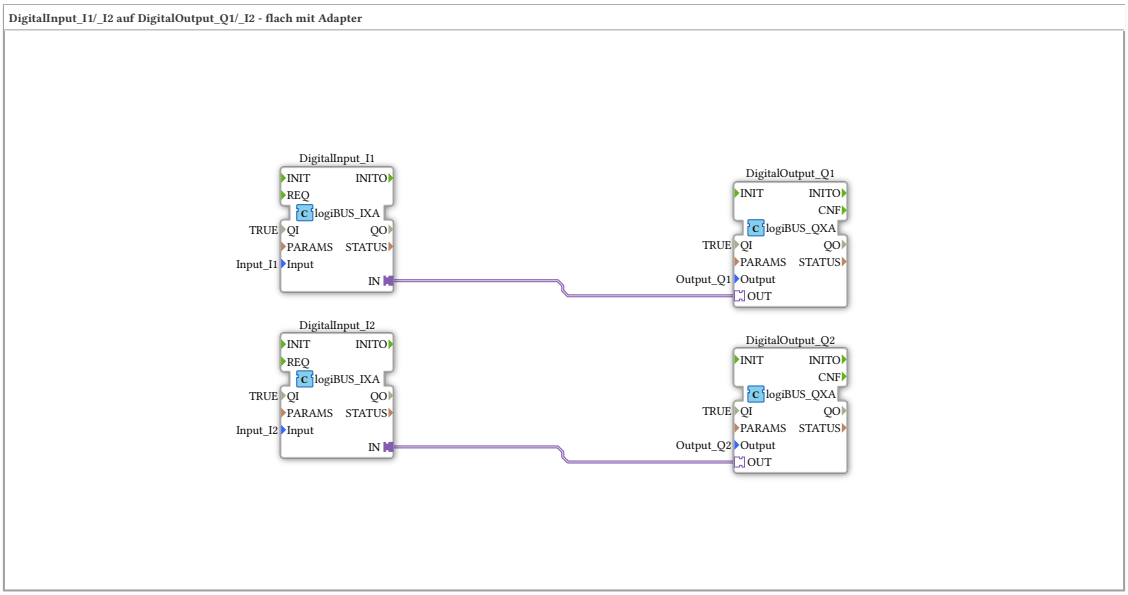
Das Hauptziel dieser Übung ist es, die parallele Verarbeitung von Signalen in der IEC 61499 zu demonstrieren. Anders als in sequenziellen Programmiermodellen (wie z.B. klassischen SPS-Zyklen in AWL), arbeiten die Funktionsbausteine in 4diac ereignisbasiert und unabhängig voneinander. Dies ermöglicht es, mehrere Steuerungsaufgaben gleichzeitig und ohne gegenseitige Beeinflussung in einer einzigen Subapplikation abzubilden.

Beschreibung und Komponenten

Die Subapplikation `Uebung_003_AX.SUB` definiert zwei separate „Stränge“ der Signalverarbeitung, die parallel existieren.

Funktionsbausteine (FBs)

Es werden zwei Paare von Ein- und Ausgangsbausteinen verwendet:



- **`DigitalInput_I1` & `DigitalOutput_Q1`**: Das erste Paar (Kanal 1). Verbindet Hardware-Eingang `I1` mit Hardware-Ausgang `Q1`.
- **`DigitalInput_I2` & `DigitalOutput_Q2`**: Das zweite Paar (Kanal 2). Verbindet Hardware-Eingang `I2` mit Hardware-Ausgang `Q2`.

Adapter-Schnittstelle: `AX.adp`

Beide Verbindungen nutzen die standardisierte Adapter-Schnittstelle `AX` für die Kommunikation.

Funktionsweise

Die Unabhängigkeit der beiden Kanäle wird durch die getrennten Adapter-Verbindungen in der Subapplikation `Uebung_003_AX.SUB` sichergestellt:

```
<AdapterConnections>
    <Connection Source="DigitalInput_I1.IN" Destination="DigitalOutput_Q1.OUT"/>
    <Connection Source="DigitalInput_I2.IN" Destination="DigitalOutput_Q2.OUT"/>
</AdapterConnections>
```

Der funktionale Ablauf:

1. Ändert sich der Zustand von `I1`, sendet `DigitalInput_I1` ein Ereignis direkt an `DigitalOutput_Q1`. Der Ausgang `Q1` schaltet.
2. Ändert sich der Zustand von `I2`, sendet `DigitalInput_I2` ein Ereignis direkt an `DigitalOutput_Q2`. Der Ausgang `Q2` schaltet.

Diese beiden Prozesse laufen völlig asynchron ab. Eine hohe Schaltfrequenz auf Kanal 1 hat keinen Einfluss auf die Reaktionszeit oder Funktion von Kanal 2.

Anwendungsbeispiel

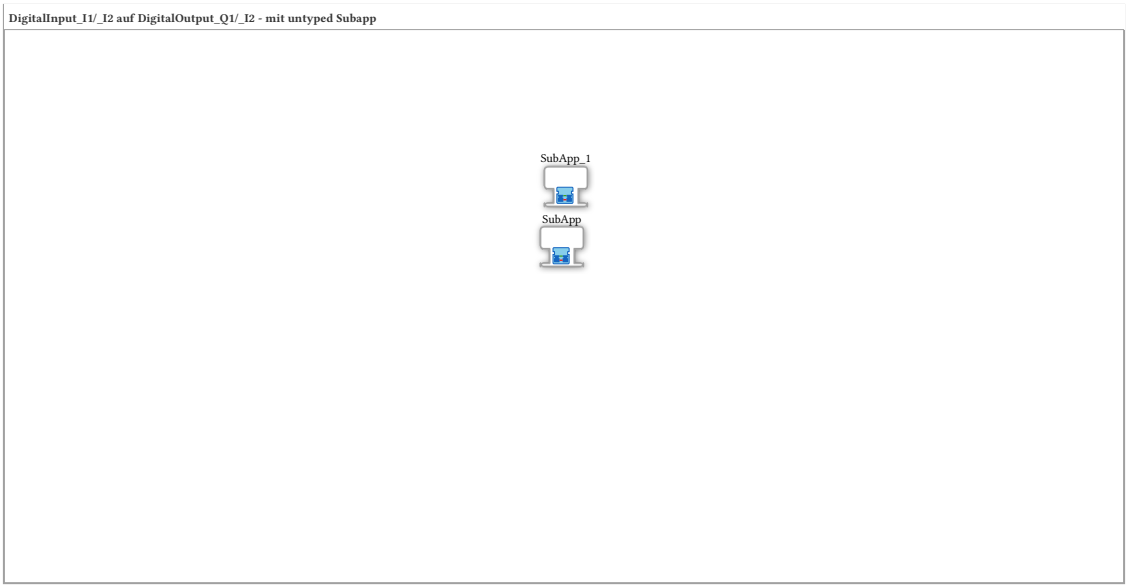
Ein einfaches Anwendungsbeispiel ist die **Steuerung von zwei unabhängigen Pumpen**:

In einem Pumpwerk gibt es zwei identische Pumpen, die jeweils über einen eigenen Vor-Ort-Schalter bedient werden. Schalter 1 (`I1`) startet Pumpe 1 (`Q1`), und Schalter 2 (`I2`) startet Pumpe 2 (`Q2`). Obwohl beide Steuerungen in demselben Steuerungsprogramm laufen, operieren sie logisch völlig getrennt voneinander. Fällt ein Sensor aus, bleibt der andere Kreis voll funktionsfähig.

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

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

2.1.2.28 Uebung_003a0_AX: DigitalInput_I1/_I2 auf DigitalOutput_Q1/_I2 - mit untyped Subapp



NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_003a0_AX`. Im Gegensatz zur Verwendung von Typen (wie in `Uebung_003a_AX`) wird hier gezeigt, wie man Logik visuell gruppieren kann, ohne separate Typ-Definitionen anzulegen. Dies geschieht durch sogenannte „Untyped SubApps“.

Ziel der Übung

Das Hauptziel dieser Übung ist es, Möglichkeiten zur rein visuellen Strukturierung von Anwendungen aufzuzeigen. Untypisierte Sub-Applikationen dienen als „Container“ oder Ordner innerhalb eines Netzwerks, um zusammengehörige Funktionen zu kapseln. Sie helfen, komplexe Diagramme aufzuräumen („Aufräumen durch Einklappen“), ohne dass man sich Gedanken über Wiederverwendbarkeit oder Schnittstellen-Definitionen machen muss.

Beschreibung und Komponenten

Die Subapplikation `Uebung_003a0_AX.SUB` enthält zwei eingebettete Sub-Applikationen (`SubApp` und `SubApp_1`), die jeweils einen eigenen Signalpfad beinhalten.

Untypisierte Sub-Applikationen

Anders als bei typisierten SubApps, die auf einer externen Definition basieren, existiert die Logik dieser Container nur innerhalb dieser spezifischen Instanz.

1. **Container `SubApp`**:
   - Beinhaltet die Logik für Kanal 2.
   - Intern: `DigitalInput_I2` verbunden mit `DigitalOutput_Q2`.
2. **Container `SubApp_1`**:
   - Beinhaltet die Logik für Kanal 1.
   - Intern: `DigitalInput_I1` verbunden mit `DigitalOutput_Q1`.

Funktionsbausteine (FBs)

Innerhalb der Container werden die bekannten Bausteine verwendet:

- **`logiBUS_IXA`**: Zum Einlesen der Signale.
- **`logiBUS_QXA`**: Zum Ausgeben der Signale.

Funktionsweise

Die Logik ist identisch zur parallelen Steuerung, jedoch ist die Ansicht hierarchisch gegliedert. Der Aufbau in `Uebung_003a0_AX.SUB` zeigt die Verschachtelung:

```
<SubAppNetwork>
    <SubApp Name="SubApp">
        <SubAppNetwork>
            <FB Name="DigitalInput_I2" ... />
            <FB Name="DigitalOutput_Q2" ... />
            <AdapterConnections>
                <Connection Source="DigitalInput_I2.IN"
Destination="DigitalOutput_Q2.OUT"/>
            </AdapterConnections>
        </SubAppNetwork>
    </SubApp>

    <SubApp Name="SubApp_1">
        <SubAppNetwork>
            <FB Name="DigitalInput_I1" ... />
            <FB Name="DigitalOutput_Q1" ... />
            <AdapterConnections>
                <Connection Source="DigitalInput_I1.IN"
Destination="DigitalOutput_Q1.OUT"/>
            </AdapterConnections>
        </SubAppNetwork>
    </SubApp>
</SubAppNetwork>
```

Der funktionale Ablauf: Die Kapselung hat keinen Einfluss auf die Laufzeit-Ausführung. Die Bausteine verhalten sich exakt so, als wären sie alle auf der obersten Ebene platziert.

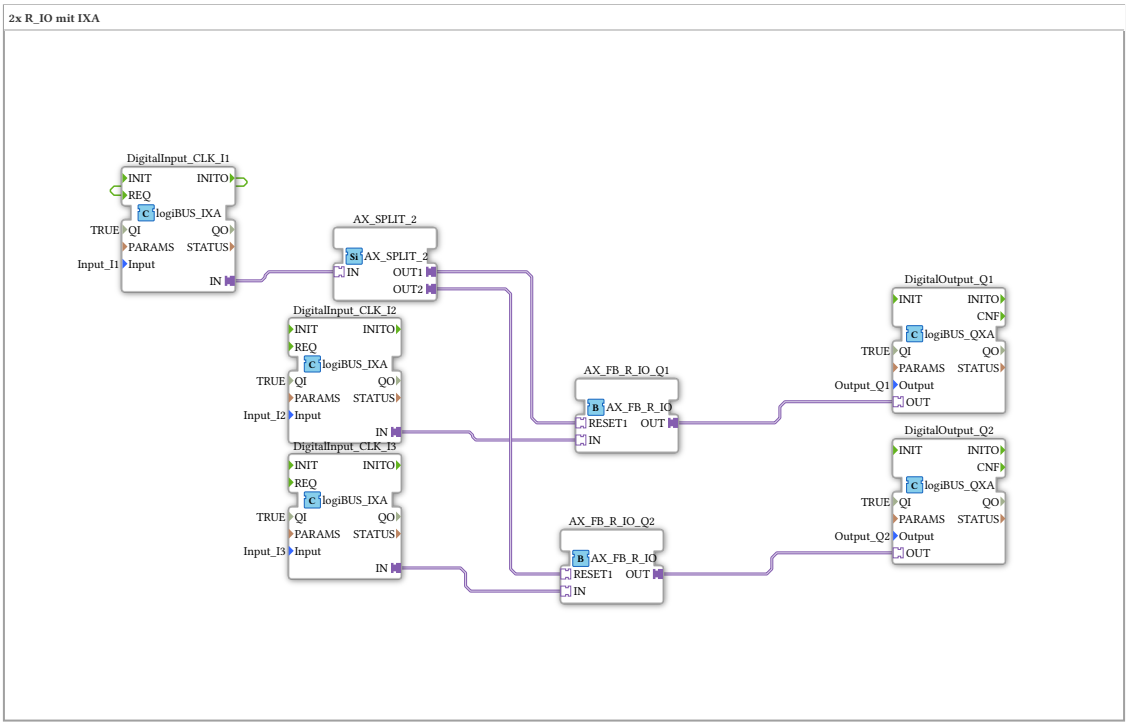
1. `SubApp_1` verarbeitet das Signal von `I1` zu `Q1`.
2. `SubApp` verarbeitet das Signal von `I2` zu `Q2`.

Anwendungsbeispiel

Ein typisches Anwendungsbeispiel ist die **Visuelle Gruppierung von Funktionsbereichen**:

In einer großen Anlage könnte man untypisierte SubApps nutzen, um die Logik thematisch zu sortieren, z.B. einen Container „Temperaturregelung“, einen „Antriebssteuerung“ und einen „Sicherheitsüberwachung“. Wenn man die Hauptansicht öffnet, sieht man nur diese drei Blöcke und nicht hunderte von einzelnen Gattern und Verbindungen. Dies erleichtert die Navigation („Zoom-In“-Effekt), ist aber nicht für die Wiederverwendung gedacht (dafür nutzt man typisierte SubApps).

2.1.2.29 Uebung_003a2b_AX: 2x R_IO mit IXA



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert die Steuerung von zwei digitalen Ausgängen (Q1 und Q2) über jeweils einen Rücksetz-Set-Funktionsbaustein (`AX_FB_R_IO`). Die Rücksetzeingänge beider Bausteine werden gemeinsam über den ersten Digitaleingang (I1) gesteuert, der als „Hausmeister-Aus“ (alles Aus mit einem Schalter) fungiert. Ein `AX_SPLIT_2` verteilt das Rücksetzsignal auf beide Kanäle. Gesetzt werden die Ausgänge über I2 (für Q1) und I3 (für Q2).

Ziel ist es, das Zusammenspiel von monostabilen Elementen (R_IO) mit Hardware-Ein- und Ausgängen sowie die Realisierung einer gemeinsamen Rücksetzfunktion zu verstehen.

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_CLK_I1**: logiBUS Digitaleingang für das gemeinsame Reset („Hausmeister-Aus“)
  - **Typ**: `logiBUS::io::DI::logiBUS_IXA`
  - **Parameter**: `QI = TRUE`, `PARAMS = ""` (beide in der Darstellung ausgeblendet), `Input = Input_I1`
  - **Erklärung**: Koppelt den physischen Eingang `I1` an die Adapterschnittstelle `IN`. Zusätzlich ist `INITO` mit `REQ` verbunden (EventConnection).
- **DigitalInput_CLK_I2**: logiBUS Digitaleingang zum Setzen von Q1
  - **Typ**: `logiBUS::io::DI::logiBUS_IXA`
  - **Parameter**: `QI = TRUE`, `PARAMS = ""` (beide ausgeblendet), `Input = Input_I2`
  - **Erklärung**: Koppelt den physischen Eingang `I2` an die Adapterschnittstelle `IN`.
- **DigitalInput_CLK_I3**: logiBUS Digitaleingang zum Setzen von Q2
  - **Typ**: `logiBUS::io::DI::logiBUS_IXA`
  - **Parameter**: `QI = TRUE`, `PARAMS = ""` (beide ausgeblendet), `Input = Input_I3`
  - **Erklärung**: Koppelt den physischen Eingang `I3` an die Adapterschnittstelle `IN`.

- **AX_SPLIT_2**: Signalverteiler
  - **Typ**: `adapter::events::unidirectional::AX_SPLIT_2`
  - **Parameter**: keine
  - **Erklärung**: Verteilt das Signal von `I1` unverändert auf die zwei Ausgänge `OUT1` und `OUT2`.
- **AX_FB_R_IO_Q1**: Rücksetz-Set-Element für Q1
  - **Typ**: `adapter::monostableElements::AX_FB_R_IO`
  - **Parameter**: keine
  - **Erklärung**: `IN` (von `I2`) setzt den Ausgang, `RESET1` (von `AX_SPLIT_2.OUT1`) setzt ihn zurück. Der Ausgang `OUT` führt zu `DigitalOutput_Q1`.
- **AX_FB_R_IO_Q2**: Rücksetz-Set-Element für Q2
  - **Typ**: `adapter::monostableElements::AX_FB_R_IO`
  - **Parameter**: keine
  - **Erklärung**: `IN` (von `I3`) setzt den Ausgang, `RESET1` (von `AX_SPLIT_2.OUT2`) setzt ihn zurück. Der Ausgang `OUT` führt zu `DigitalOutput_Q2`.
- **DigitalOutput_Q1**: logiBUS Digitalausgang für Q1
  - **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
  - **Parameter**: `QI = TRUE`, `Output = Output_Q1`
  - **Erklärung**: Gibt den Zustand von `AX_FB_R_IO_Q1.OUT` an den physischen Ausgang `Q1` weiter.
- **DigitalOutput_Q2**: logiBUS Digitalausgang für Q2
  - **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
  - **Parameter**: `QI = TRUE`, `Output = Output_Q2`
  - **Erklärung**: Gibt den Zustand von `AX_FB_R_IO_Q2.OUT` an den physischen Ausgang `Q2` weiter.

Programmablauf und Verbindungen

1. `DigitalInput_CLK_I1.INITO` → `DigitalInput_CLK_I1.REQ` (EventConnection): Der Baustein fordert nach der Initialisierung selbst den Eingangswert an.
2. `DigitalInput_CLK_I1.IN` → `AX_SPLIT_2.IN` (AdapterConnection): Das Signal von `I1` wird dem Verteiler zugeführt.
3. `AX_SPLIT_2.OUT1` → `AX_FB_R_IO_Q1.RESET1` und `AX_SPLIT_2.OUT2` → `AX_FB_R_IO_Q2.RESET1`: `I1` wirkt als gemeinsames Reset für beide Kanäle.
4. `DigitalInput_CLK_I2.IN` → `AX_FB_R_IO_Q1.IN`: `I2` setzt Q1.
5. `DigitalInput_CLK_I3.IN` → `AX_FB_R_IO_Q2.IN`: `I3` setzt Q2.
6. `AX_FB_R_IO_Q1.OUT` → `DigitalOutput_Q1.OUT` und `AX_FB_R_IO_Q2.OUT` → `DigitalOutput_Q2.OUT`: Die Zustände werden auf die physischen Ausgänge `Q1` bzw. `Q2` gelegt.

Hinweise aus den Kommentaren im Netzwerk:

- Hausmeister-Aus (alles Aus mit einem Schalter): Der Schalter müsste RASTEND sein. Ansonsten sind die Ausgänge nur so lange aus, wie der Taster gehalten wird.
- Vorteil dieser Lösung: Wird an `RESET1` nichts angeschlossen, ist der Baustein funktionsfähig. Das Prinzip entspricht dem `FB_RS`.
- Ein AND wäre auch möglich, dann wäre der Hausmeister ein Zustimmschalter.

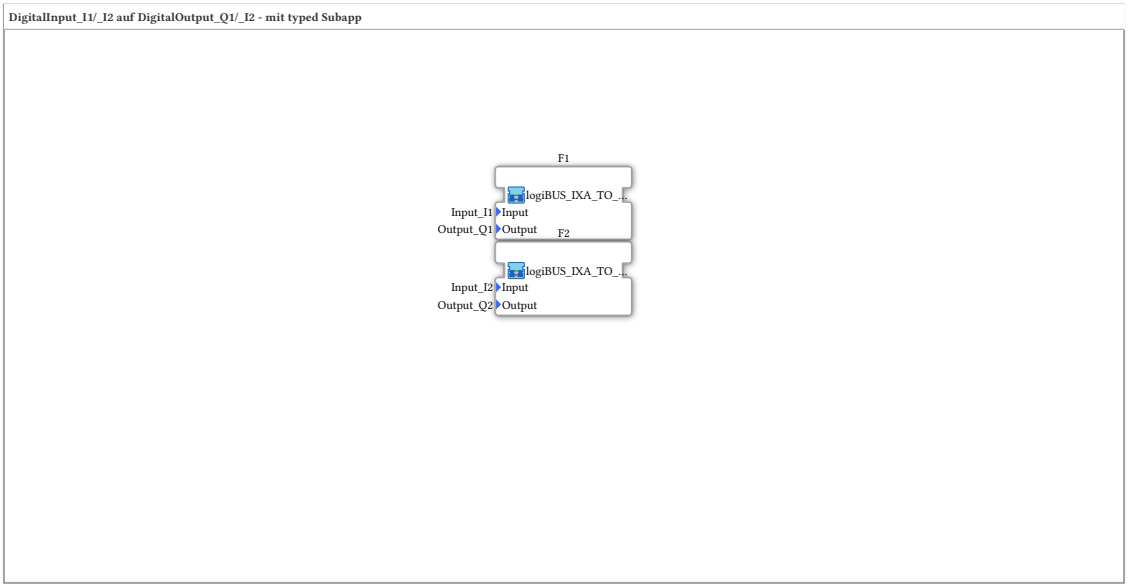
Zusammenfassung

Die Übung zeigt eine Schaltung zur Steuerung zweier Ausgänge mit einem gemeinsamen Rücksetzsignal: `I2` setzt `Q1`, `I3` setzt `Q2`, und `I1` setzt über `AX_SPLIT_2` beide `AX_FB_R_IO`-Bausteine zurück („Hausmeister-Aus“). Die Aufteilung des Rücksetzsignals über `AX_SPLIT_2` macht die Schaltung übersichtlich und erweiterbar.

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

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

2.1.2.30 Uebung_003a_AX: DigitalInput_I1/_I2 auf DigitalOutput_Q1/_I2 - mit typed Subapp





Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_003a_AX`. Die Übung demonstriert einen fortgeschrittenen Ansatz zur Strukturierung von IEC 61499-Anwendungen: die Verwendung von typisierten Sub-Applikationen („Typed SubApps“) zur Kapselung und Wiederverwendung von Logik.

Ziel der Übung

Das Hauptziel ist es zu zeigen, wie redundanter Code vermieden werden kann. Anstatt identische Strukturen (z.B. die Verbindung eines Eingangs mit einem Ausgang) mehrfach zu zeichnen, wird einmalig ein generischer Baustein definiert. Dieser kann dann beliebig oft instanziiert und konfiguriert werden. Dies erhöht die Übersichtlichkeit und Wartbarkeit von großen Projekten erheblich.

Beschreibung und Komponenten

Die Subapplikation `Uebung_003a_AX.SUB` verwendet zwei Instanzen eines selbst definierten Sub-Typs, um zwei Signalpfade zu realisieren.

Typisierte Sub-Applikation: `Uebung_003a_AX_sub`

Dieser Baustein kapselt die grundlegende Logik: „Lies einen Eingang und schreibe auf einen Ausgang“. Er verfügt über Schnittstellen zur Parametrierung:

- **`Input`**: Bestimmt, welcher physische Eingang gelesen werden soll (z.B. `Input_I1`).
- **`Output`**: Bestimmt, welcher physische Ausgang geschaltet werden soll (z.B. `Output_Q1`).

Intern enthält dieser Sub-Typ:

- Einen `logiBUS_IXA` Baustein zum Lesen des Eingangs.
- Einen `logiBUS_QXA` Baustein zum Schreiben des Ausgangs.
- Eine Adapter-Verbindung, die beide direkt verknüpft.

Instanzen in der Hauptanwendung

In `Uebung_003a_AX` werden zwei Instanzen dieses Typs erzeugt:

- **`F1`**: Konfiguriert für `Input_I1` auf `Output_Q1`.
- **`F2`**: Konfiguriert für `Input_I2` auf `Output_Q2`.

Funktionsweise

Die Logik ist im Inneren der Sub-Applikation versteckt („Information Hiding“). Die Hauptanwendung definiert nur noch die Verschaltung der Parameter. Der Aufbau in `Uebung_003a_AX.SUB` ist daher extrem kompakt:

```
<SubApp Name="F1" Type="Uebungen::Uebung_003a_AX_sub">
    <Parameter Name="Input" Value="Input_I1"/>
    <Parameter Name="Output" Value="Output_Q1"/>
</SubApp>
<SubApp Name="F2" Type="Uebungen::Uebung_003a_AX_sub">
    <Parameter Name="Input" Value="Input_I2"/>
    <Parameter Name="Output" Value="Output_Q2"/>
</SubApp>
```

Der funktionale Ablauf entspricht exakt dem der `Uebung_003_AX` (parallele Steuerung), jedoch ist die Implementierung modularer. Jede Instanz (`F1`, `F2`) arbeitet als eigenständiger, isolierter Block, der seine interne Adapter-Logik ausführt.

Anwendungsbeispiel

Ein perfektes Anwendungsbeispiel ist die **Objektorientierte Anlagensteuerung**:

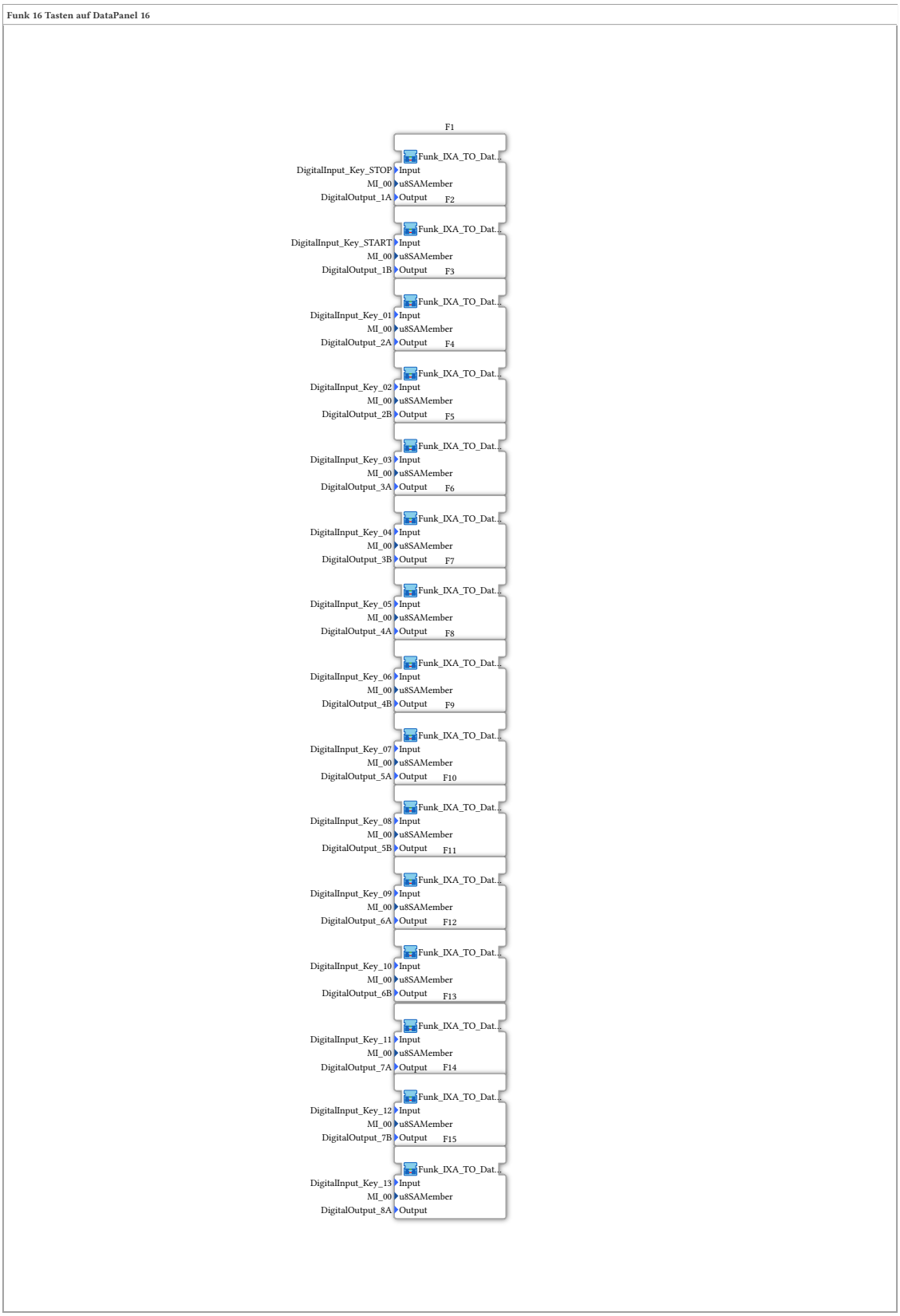
Stellen Sie sich eine Förderbandanlage mit 50 identischen Förderbändern vor. Jedes Band hat einen Motor (Ausgang) und eine Lichtschranke (Eingang). Anstatt 50 mal die gleichen Bausteine und Verbindungen zu zeichnen, erstellt man einmal einen Typ „Förderband-Modul“. In der Hauptanwendung platziert man dann einfach 50 Instanzen dieses Moduls und weist ihnen nur noch die physikalischen Adressen zu. Ändert sich später die Logik (z.B. soll der Motor verzögert stoppen), muss man dies nur an einer Stelle (im Typ) ändern, und alle 50 Bänder werden automatisch aktualisiert.

2.1.2.31 Uebung_003a_AX_sub

Hinweis

Diese Übung existiert in den aktuellen Trainingsquellen nicht mehr (entfernt, zusammengelegt oder umbenannt). Diese Seite bleibt nur als Platzhalter für bestehende Verweise erhalten. Die aktuellen Übungen stehen in der Übungsübersicht.

2.1.2.32 Uebung_003b2_AX: Funk 16 Tasten auf DataPanel 16



NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_003b2_AX`. Diese Übung zeigt die Skalierbarkeit des Systems am Beispiel der Verknüpfung von 15 Funktasten mit 15 digitalen Ausgängen, realisiert durch untypisierte Sub-Applikationen (bzw. wiederverwendete typisierte Sub-Applikationen, die hier manuell platziert wurden).

Ziel der Übung

Das Ziel dieser Übung ist es, eine große Anzahl von gleichartigen Verbindungen effizient zu verwalten. Anstatt 15 Mal die gleichen Funktionsbausteine und Verbindungen in der Hauptansicht zu zeichnen, wird die Logik in modulare Blöcke (Sub-Applikationen) verpackt. Dies demonstriert, wie IEC 61499 für umfangreiche I/O-Mappings eingesetzt wird.

Beschreibung und Komponenten

Die Subapplikation `Uebung_003b2_AX.SUB` instanziiert 15 mal den Sub-Typ `Uebung_003b2_sub_AX` (hier als `F1` bis `F15` benannt).

Sub-Komponente: `Uebung_003b2_sub_AX`

Dieser Sub-Baustein verbindet einen Funkeingang (`Funk_IXA`) mit einem DataPanel-Ausgang (`DataPanel_MI_QXA`). Er verfügt über Parameter zur Konfiguration:

- **`Input`**: Der Funk-Taster (z.B. `DigitalInput_Key_START`).
- **`Output`**: Der physische Ausgang (z.B. `DigitalOutput_1B`).
- **`u8SAMember`**: Die Adresse des DataPanels.

Mapping

Die Übung mappt folgende Signale:

- `STOP` -> (nicht belegt/intern)
- `START` -> `F2`
- `Key_01` bis `Key_13` -> `F3` bis `F15`

(Hinweis: Die genaue Zuordnung der Parameter ist in der `.SUB`-Datei für `Uebung_003b2_AX` im Gegensatz zu `Uebung_003b3_AX` nicht explizit sichtbar, da hier untypisierte oder eingebettete SubApps verwendet wurden, bei denen die Parameter im Inneren gesetzt sind oder Default-Werte nutzen. In `Uebung_003b3_AX` sind die Parameter explizit nach außen geführt.)

Funktionsweise

Jede Instanz `F1` bis `F15` arbeitet unabhängig:

1. Der `Funk_IXA`-Baustein empfängt das Signal der entsprechenden Taste von der Funkfernbedienung.
2. Das Signal wird über eine Adapter-Verbindung an den `DataPanel_MI_QXA`-Baustein weitergeleitet.
3. Der Ausgang am DataPanel schaltet entsprechend.

Da alle 15 Kanäle parallel arbeiten, können mehrere Tasten gleichzeitig gedrückt werden, und die entsprechenden Ausgänge reagieren simultan.

Anwendungsbeispiel

Ein Anwendungsbeispiel ist ein **Bedienfeld für einen Ladekran**:

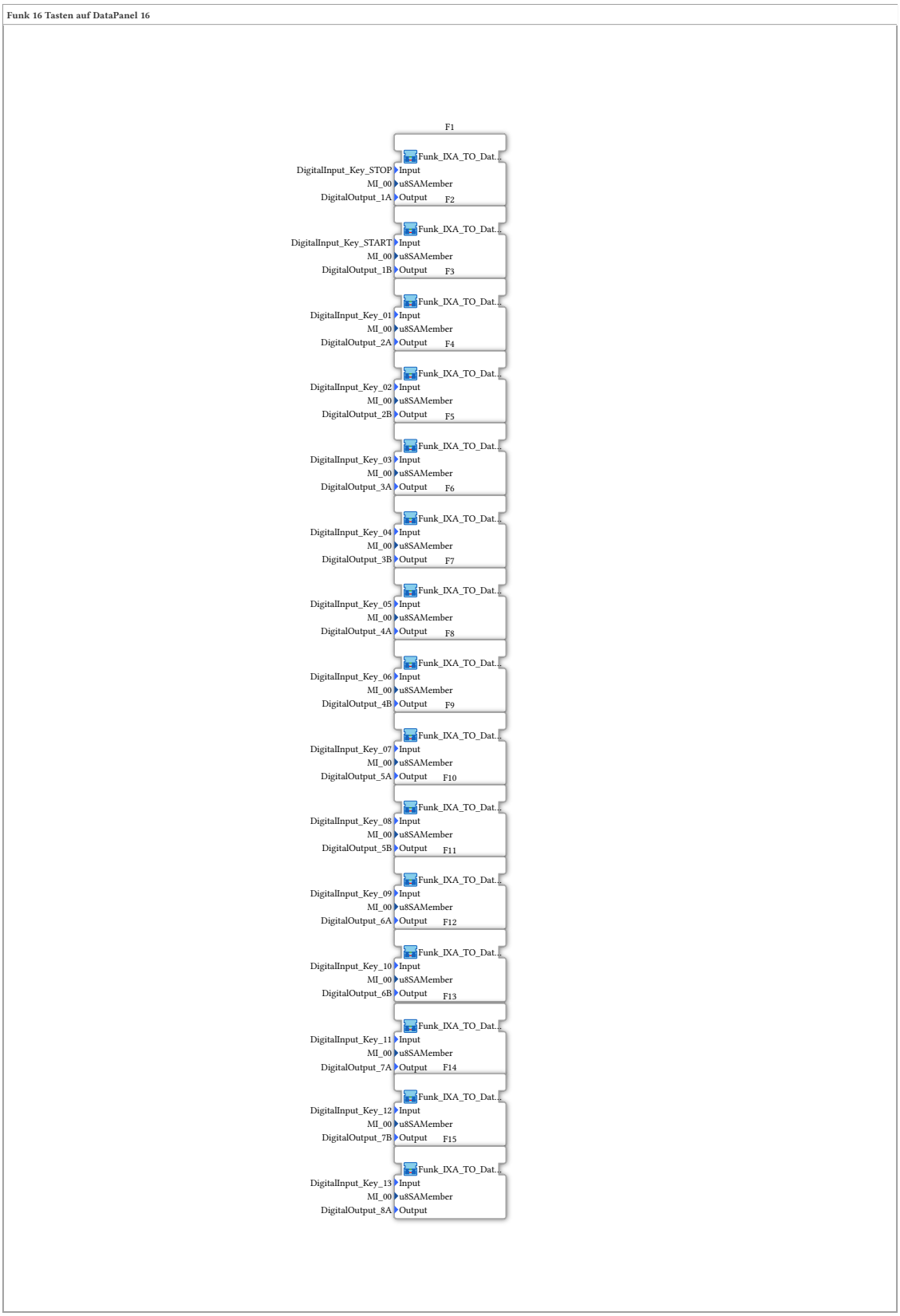
Der Kranführer hat eine Funkfernbedienung mit vielen Tasten (Heben, Senken, Drehen, Teleskopieren, etc.). Jede Taste muss ein spezifisches Hydraulikventil ansteuern. Durch die Verwendung von Sub-Applikationen bleibt der Schaltplan übersichtlich („Eine Kiste pro Funktion“), obwohl im Hintergrund 30 oder mehr Bausteine aktiv sind.

2.1.2.33 Uebung_003b2_sub_AX

Hinweis

Diese Übung existiert nicht mehr als eigene Übung. Der Baustein wurde in die Bibliothek verschoben und heißt jetzt Funk_IXA_TO_DataPanel_MI_QXA. Beschreibung, Schnittstelle und Netzwerk stehen dort.

2.1.2.34 Uebung_003b3_AX: Funk 16 Tasten auf DataPanel 16



NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_003b3_AX`. Diese Übung ist die konsequente Weiterentwicklung von `Uebung_003b2_AX`. Sie nutzt explizit parametrierte typisierte Sub-Applikationen, um 15 Funktasten auf 15 Ausgänge zu mappen.

Ziel der Übung

Das Ziel ist maximale Wiederverwendbarkeit und Übersichtlichkeit. Durch die Verwendung von typisierten Sub-Applikationen mit exponierten Parametern kann die gesamte Konfiguration (welche Taste auf welchen Ausgang) direkt in der Hauptansicht vorgenommen werden, ohne in die Sub-Bausteine hineinklicken zu müssen.

Beschreibung und Komponenten

Die Subapplikation `Uebung_003b3_AX.SUB` instanziiert 15 mal den Typ `Uebung_003b3_sub_AX`.

Typisierte Sub-Applikation: `Uebung_003b3_sub_AX`

Dieser Baustein ist identisch zur Funktionalität in `Uebung_003b2_sub_AX`, jedoch sind die Parameter `Input`, `Output` und `u8SAMember` sauber herausgeführt.

Konfiguration der Instanzen

Die Zuweisung ist in der Datei klar lesbar:

- **`F1`**: `STOP` -> `Output_1A` (Achtung: Logik prüfen, ob STOP ggf. invertiert oder speziell behandelt wird)
- **`F2`**: `START` -> `Output_1B`
- **`F3`**: `Key_01` -> `Output_2A`
- **`F4`**: `Key_02` -> `Output_2B`
- ...
- **`F15`**: `Key_13` -> `Output_8A`

Alle Instanzen nutzen `MI_00` als `u8SAMember` (Adresse des DataPanels).

Funktionsweise

Die Funktionsweise ist identisch zur vorherigen Übung:

1. Die Hauptanwendung übergibt die Parameter (z.B. „Benutze Taste 1“) an die Sub-Applikation.
2. Die Sub-Applikation konfiguriert ihren internen `Funk_IXA` Baustein entsprechend.
3. Das Signal wird gelesen und via Adapter an den `DataPanel_MI_QXA` weitergeleitet.
4. Der Ausgang schaltet.

Der Vorteil hier ist die Wartbarkeit: Soll Taste 1 plötzlich Ausgang 8A schalten, muss man nur den Parameter an `F3` ändern, anstatt Leitungen neu zu ziehen.

Anwendungsbeispiel

Ein Anwendungsbeispiel ist die **Serienfertigung von Sondermaschinen**:

Eine Firma baut Müllpressen, die mal mit 4, mal mit 8 oder 12 Tasten bestückt sind. Die Software ist immer gleich aufgebaut: Man nimmt den Baustein „Tasten-Kanal“ und platziert ihn so oft wie nötig. Die Zuordnung, welcher Hydraulikzylinder zu welcher Taste gehört, geschieht rein über Parameterlisten. Das reduziert Fehler und beschleunigt die Inbetriebnahme enorm.

2.1.2.35 Uebung_003b3_sub_AX

Hinweis

Diese Übung existiert in den aktuellen Trainingsquellen nicht mehr (entfernt, zusammengelegt oder umbenannt). Diese Seite bleibt nur als Platzhalter für bestehende Verweise erhalten. Die aktuellen Übungen stehen in der Übungsübersicht.

2.1.2.36 Uebung_003b_AX: DigitalInput_I1-2-3-4 auf DigitalOutput_Q1-2-3-4

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_003b_AX (DigitalInput_I1-2-3-4 auf DigitalOutput_Q1-2-3-4).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **DigitalInput_I1-2-3-4 auf DigitalOutput_Q1-2-3-4**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_003b_AX.SUB, welche die folgende Bausteinstruktur verwendet:

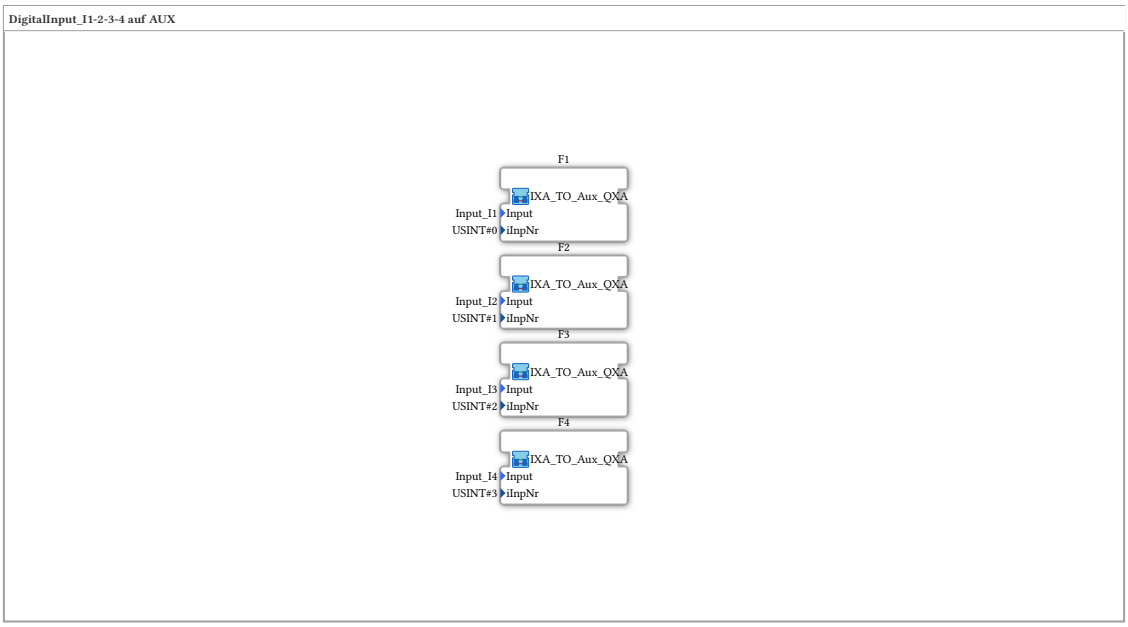
Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Signalweiterleitung**: Die Bausteine verarbeiten Daten- und Steuersignale gemäß ihrer Konfiguration.

Zusammenfassung

Die Übung Uebung_003b_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.37 Uebung_003c_AX: DigitalInput_I1-2-3-4 auf AUX



NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_003c_AX`. Diese Übung demonstriert die Anbindung von lokalen digitalen Eingängen an das ISOBUS-System als „Auxiliary Inputs“. Dies ermöglicht die Nutzung lokaler Schalter zur Steuerung von ISOBUS-Geräten.

Ziel der Übung

Das Ziel ist es, lokale Hardware-Eingänge (`I1` bis `I4`) für das ISOBUS-Netzwerk verfügbar zu machen. In der ISOBUS-Welt können Eingabegeräte (Joysticks, Schalterfelder) als „Auxiliary Inputs“ definiert werden, die dann vom Benutzer flexibel auf Maschinenfunktionen gemappt werden können.

Beschreibung und Komponenten

Die Subapplikation `Uebung_003c_AX.SUB` nutzt untypisierte (bzw. wiederverwendete typisierte) Sub-Applikationen, um vier Kanäle zu definieren.

Sub-Komponente: `Uebung_003c_sub_AX`

Dieser Sub-Baustein verbindet einen lokalen digitalen Eingang mit einem ISOBUS-Auxiliary-Ausgangsbaustein.

- **Eingang**: `logiBUS_IXA` (Liest den lokalen Pin)
- **Ausgang**: `Aux_QXA` (Sendet den Status als ISOBUS AUX Nachricht)
- **Parameter**:
  - `Input`: Der lokale Eingang (z.B. `Input_I1`).
  - `iInpNr`: Die Nummer des AUX-Inputs im ISOBUS-Pool (Index 0 bis n).

Konfiguration

Die Übung mappt vier Eingänge:

- `F1`: `Input_I1` -> AUX Index 0
- `F2`: `Input_I2` -> AUX Index 1 (implizit, Parameter-Mapping in der .SUB prüfen)
- `F3`: `Input_I3` -> AUX Index 2
- `F4`: `Input_I4` -> AUX Index 3

(Hinweis: In der bereitgestellten .SUB-Datei sind die Parameter `Input` und `iInpNr` nicht explizit bei den Instanzen `F1`...`F4` sichtbar, was typisch für untypisierte SubApps ist, wo die Werte im Inneren gesetzt sind. Bei einer sauberen typisierten Lösung wären diese Werte wie in `Uebung_003b3_AX` sichtbar.)

Funktionsweise

1. Der Bediener betätigt Schalter `I1`.
2. Der `logiBUS_IXA`-Baustein erfasst das Signal.
3. Das Signal wird via Adapter an den `Aux_QXA`-Baustein weitergeleitet.
4. Der `Aux_QXA`-Baustein generiert die entsprechende ISOBUS-Nachricht, dass der „Auxiliary Input 0“ betätigt wurde.
5. Ein angeschlossenes ISOBUS-Gerät (z.B. eine Feldspritze) empfängt dies und führt die Funktion aus, die der Benutzer diesem Schalter zugewiesen hat.

Anwendungsbeispiel

Ein **Nachrüst-Joystick**:

Ein älterer Traktor hat keinen ISOBUS-Joystick. Man installiert vier einfache Taster am Armaturenbrett und verbindet sie mit der Steuerung. Durch diese Software-Logik tauchen diese vier Taster plötzlich als vollwertige ISOBUS-Bedienelemente auf dem Terminal auf und können genutzt werden, um modernste Anbaugeräte zu steuern.

2.1.2.38 Uebung_003c_sub_AX

Hinweis

Diese Übung existiert nicht mehr als eigene Übung. Der Baustein wurde in die Bibliothek verschoben und heißt jetzt IXA_TO_Aux_QXA. Beschreibung, Schnittstelle und Netzwerk stehen dort.

2.1.2.39 Uebung_003d_AX: DigitalInput_I1/_I2 auf DigitalOutput_Q1/_I2



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_003d_AX`. Diese Übung ist strukturell nahezu identisch mit `Uebung_003_AX` und dient der Festigung des Verständnisses für parallele Signalpfade.

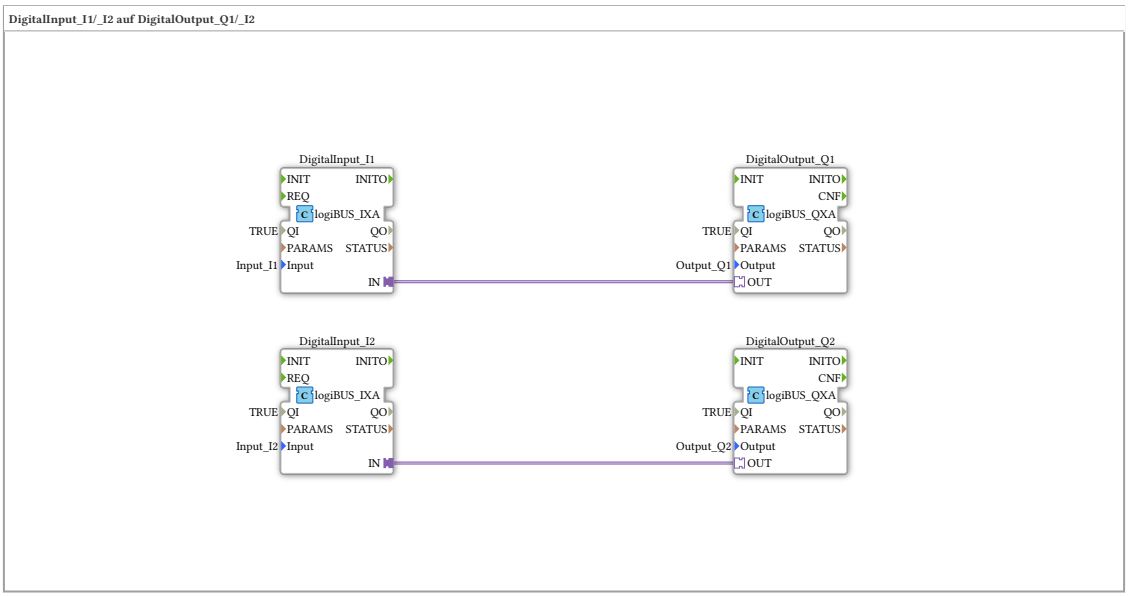
Ziel der Übung

Das Ziel ist die Wiederholung der direkten I/O-Verknüpfung mittels Adapter-Technologie. Es wird sichergestellt, dass das Konzept der unabhängigen Datenflüsse verstanden wurde.

Beschreibung und Komponenten

Die Subapplikation `Uebung_003d_AX.SUB` verbindet zwei Eingänge mit zwei Ausgängen.

Funktionsbausteine (FBs)



- **`DigitalInput_I1`** -> **`DigitalOutput_Q1`**
- **`DigitalInput_I2`** -> **`DigitalOutput_Q2`**

Die Bausteintypen sind `logiBUS_IXA` und `logiBUS_QXA`, verbunden über den Adapter `AX`.

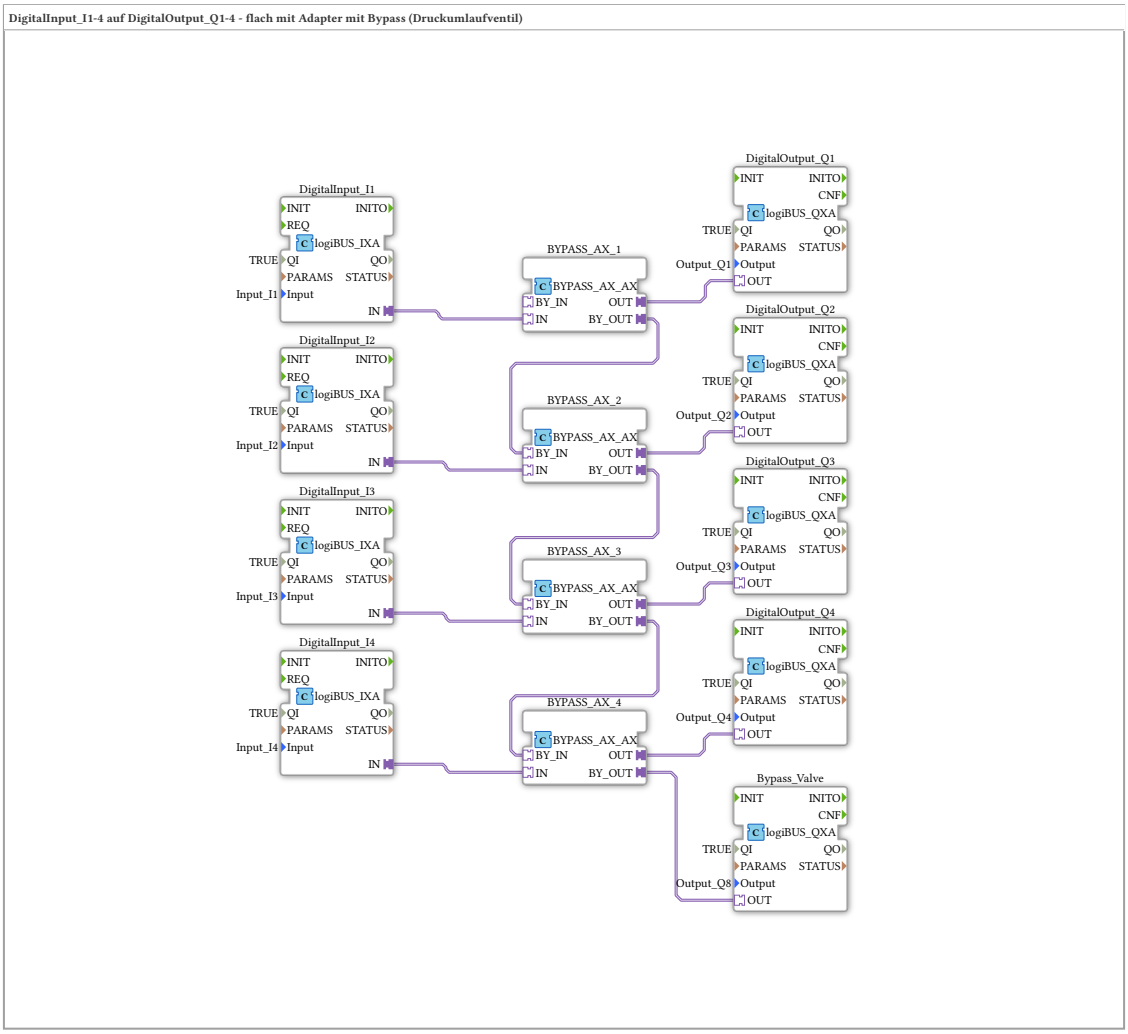
Funktionsweise

Siehe `Uebung_003_AX`. Die Signale werden 1:1 und latenzarm von den Eingängen auf die Ausgänge durchgeschleift.

Anwendungsbeispiel

Diese Übung kann als Template für **einfache Verdrahtungstests** genutzt werden. Wenn man eine neue Steuerung in Betrieb nimmt, lädt man oft so ein „dummes“ Programm hoch, um zu prüfen, ob physikalisch alles korrekt angeschlossen ist (Schalter betätigen -> LED geht an).

2.1.2.40 Uebung_003e_AX: DigitalInput_I1-4 auf DigitalOutput_Q1-4 - flach mit Adapter mit Bypass (Druckumlaufventil)



• – ○ • – ○ • – ○ •

Einleitung

Die Übung realisiert eine einfache Durchschaltung von vier digitalen Eingängen (I1–I4) auf vier digitale Ausgänge (Q1–Q4). Zusätzlich ist eine Bypass-Funktion (Druckumlaufventil) integriert, die über ein gemeinsames Magnetventil (Output_Q8) gesteuert wird. Die Signale werden über Bypass-Adapter (BYPASS_AX_AX) geführt, die einen Haupt- und einen Bypass-Pfad bereitstellen. Durch die Kaskadierung der Bypass-Pfade kann der gesamte Signalfluss umgeleitet werden.

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine

DigitalInput_Ix (x=1..4)

- **Typ**: `logiBUS::io::DI::logiBUS_IXA`
- **Parameter**: QI = `TRUE`; Input = `Input_I1` (bzw. `_I2`, `_I3`, `_I4`)
- **Funktionsweise**: Liest den digitalen Eingangswert vom logiBUS-IO-System ein.

DigitalOutput_Qx (x=1..4) und Bypass_Valve

- **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
- **Parameter**:
  - DigitalOutput_Q1..Q4: QI = `TRUE`; Output = `Output_Q1` .. `Output_Q4`
  - Bypass_Valve: QI = `TRUE`; Output = `Output_Q8`
- **Funktionsweise**: Gibt das digitale Signal auf den entsprechenden logiBUS-Ausgang aus.

BYPASS_AX_x (x=1..4)

- **Typ**: `logiBUS::signalprocessing::bypass::BYPASS_AX_AX`
- **Parameter**: keine (reine Adapterverbindungen)
- **Ereignisausgang/-eingang**: keine
- **Datenausgang/-eingang**: Adapter-Ports `IN`, `OUT`, `BY_IN`, `BY_OUT`
- **Funktionsweise**: Der Baustein besitzt zwei Signalpfade:
  - **Hauptpfad**: `IN` ⟶ `OUT` – leitet das Eingangssignal direkt zum Ausgang.
  - **Bypass-Pfad**: `BY_IN` ⟶ `BY_OUT` – dieser Pfad wird aktiv, sobald das nachgeschaltete Bypass-Ventil öffnet. In dieser Übung sind die Bypass-Pfade kaskadiert, sodass der Bypass des ersten Blocks den des zweiten speist usw. Dadurch kann der gesamte Signalfluss über die Bypass-Kette zum gemeinsamen Ventil (Bypass_Valve) umgeleitet werden.

Programmablauf und Verbindungen

Die gesamte Verschaltung erfolgt über **Adapterverbindungen** (keine Daten- oder Ereignisverbindungen):

- **Hauptpfad**: Jeder Digitaleingang (`Input_I1` … `Input_I4`) ist über den `IN`-Port des zugehörigen `BYPASS_AX`-Blocks mit dem `OUT`-Port verbunden, der zum entsprechenden Digitalausgang (`Output_Q1` … `Output_Q4`) führt.
- **Bypass-Pfad (kaskadiert)**:

```
BYPASS_AX_1.BY_OUT → BYPASS_AX_2.BY_IN
BYPASS_AX_2.BY_OUT → BYPASS_AX_3.BY_IN
BYPASS_AX_3.BY_OUT → BYPASS_AX_4.BY_IN
BYPASS_AX_4.BY_OUT → Bypass_Valve.OUT (Output_Q8)
```

- **Funktionsweise des Bypass**: Wird das gemeinsame Bypass-Ventil (`Output_Q8`) geschaltet, öffnet der Bypass-Pfad. Das Signal wird dann nicht mehr über die Hauptausgänge (Q1–Q4) ausgegeben, sondern über die Bypass-Kette zum Ventil geleitet. Dies simuliert ein Druckumlaufventil, wie es in hydraulischen oder pneumatischen Steuerungen vorkommt.

Zusammenfassung

Die Übung vermittelt den Umgang mit **Adapterverbindungen** und **Bypass-Logik** in der 4diac-IDE. Es wird gezeigt, wie vier digitale Eingänge über Bypass-Blöcke auf Ausgänge geschaltet werden können und wie eine Kaskadierung der Bypass-Signale ein gemeinsames Magnetventil steuert. Die Verwendung der logiBUS-IO-Bausteine stellt dabei die Verbindung zu realen oder simulierten Ein-/Ausgängen her.

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

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

2.1.2.41 Uebung_003f_AX: DigitalInput_I1-8 auf DigitalOutput_Q1-8



• – ◦ • – ◦ • – ◦ •

Einleitung

Die Übung schaltet alle acht digitalen Eingänge `Input_I1` bis `Input_I8` direkt auf die acht digitalen Ausgänge `Output_Q1` bis `Output_Q8` durch. Sie entspricht Uebung_003b_AX, nutzt aber alle 8 Kanäle statt nur 4. Jeder Kanal besteht aus einer Instanz des wiederverwendbaren Bausteins `logiBUS_IX_TO_logiBUS_QX`; die Instanzen sind untereinander nicht verbunden.

Verwendete Funktionsbausteine (FBs)

- **F1 … F8**: je ein Kanal Eingang → Ausgang
  - **Typ**: `MyLib::sys::logiBUS_IX_TO_logiBUS_QX` (SubApp-Instanz)
  - **Parameter** (je Instanz):

Instanz	`Input`	`Output`
F1	`Input_I1`	`Output_Q1`
F2	`Input_I2`	`Output_Q2`
F3	`Input_I3`	`Output_Q3`
F4	`Input_I4`	`Output_Q4`
F5	`Input_I5`	`Output_Q5`
F6	`Input_I6`	`Output_Q6`
F7	`Input_I7`	`Output_Q7`
F8	`Input_I8`	`Output_Q8`

- **Erklärung**: Der Baustein enthält intern ein `logiBUS_IX` (`QI = TRUE`) und ein `logiBUS_QX` (`QI = TRUE`). Das Ereignis `IND` des Eingangs löst `REQ` am Ausgang aus, der Zustand wird zusätzlich per Datenverbindung von `IN` nach `OUT` übertragen. Die Parameter `Input` und `Output` wählen den physischen Kanal.

Programmablauf und Verbindungen

1. Auf oberster Ebene gibt es keine Verbindungen zwischen den Instanzen `F1` bis `F8`; die Subapplikation hat auch keine eigene Schnittstelle.
2. In jeder Instanz wird `Input` an das interne `logiBUS_IX` und `Output` an das interne `logiBUS_QX` übergeben.
3. Intern verbunden sind `logiBUS_IX.IND` ⟶ `logiBUS_QX.REQ` (Ereignis) und `logiBUS_IX.IN` ⟶ `logiBUS_QX.OUT` (Daten).

Verhalten: Jeder Ausgang `Q1` … `Q8` folgt dem Zustand des gleichnamigen Eingangs `I1` … `I8`.

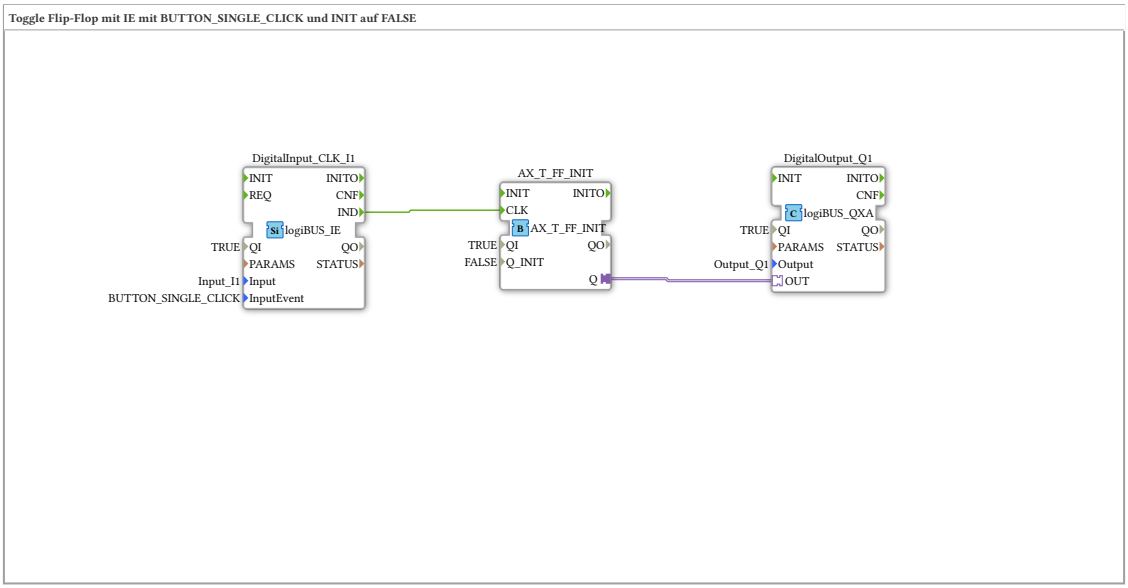
Zusammenfassung

Die Übung zeigt, wie sich durch mehrfaches Instanziieren eines parametrierbaren Bausteins mit nur zwei Parametern je Kanal acht Eingänge auf acht Ausgänge durchschalten lassen, ohne dass Verbindungen auf der obersten Ebene nötig sind.

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

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

2.1.2.42 Uebung_004a10a_AX: Toggle Flip-Flop mit IE mit BUTTON_SINGLE_CLICK und INIT auf FALSE



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung realisiert ein Toggle-Flip-Flop (T-FF) mit einem Initialwert von `FALSE`. Der Ausgangszustand wird bei jedem einfachen Tastendruck (Ereignis `BUTTON_SINGLE_CLICK`) umgeschaltet. Die Steuerung erfolgt über die logiBUS-Hardware, wobei ein digitaler Eingang (`Input_I1`) und ein digitaler Ausgang (`Output_Q1`) verwendet werden.

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine

Baustein `DigitalInput_CLK_I1`

- **Typ**: `logiBUS::io::DI::logiBUS_IE`

- **Verwendete interne FBs**: Keine (Hardwaretreiberbaustein)
- **Parameter**:
  - `QI` = `TRUE` (Aktivierung des Bausteins)
  - `Input` = `Input_I1` (physischer Eingangskanal)
  - `InputEvent` = `BUTTON_SINGLE_CLICK` (auslösendes Ereignis bei einfachem Tastendruck)
- **Funktionsweise**: Dieser Baustein erfasst den Zustand des Eingangs `Input_I1` und erzeugt bei einem Tastendruck (Ereignis vom Typ `BUTTON_SINGLE_CLICK`) ein Ereignis am Ausgang `IND`. Das Signal wird getaktet und an das nachfolgende Flip-Flop weitergegeben.

Baustein `AX_T_FF`

- **Typ**: `adapter::events::unidirectional::AX_T_FF_INIT`
- **Verwendete interne FBs**: Keine (Standard-Flip-Flop-Baustein)
- **Parameter**:
  - `QI` = `TRUE` (Aktivierung des Bausteins)
  - `Q_INIT` = `FALSE` (Anfangszustand des Ausgangs)
- **Funktionsweise**: Dies ist ein Toggle-Flip-Flop. Bei jedem empfangenen Ereignis am Eingang `CLK` (verbunden mit `IND` des Eingangsbausteins) wird der interne Zustand `Q` umgeschaltet. Der Ausgangswert wird über den Adapterausgang `Q` bereitgestellt. Der Startwert ist `FALSE`, sodass nach dem ersten Tastendruck der Zustand auf `TRUE` wechselt.

Baustein `DigitalOutput_Q1`

- **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
- **Verwendete interne FBs**: Keine (Hardwaretreiberbaustein)
- **Parameter**:
  - `QI` = `TRUE` (Aktivierung des Bausteins)
  - `Output` = `Output_Q1` (physischer Ausgangskanal)
- **Funktionsweise**: Dieser Baustein übernimmt den aktuellen Ausgangswert des Flip-Flops (über den Adapteranschluss `OUT`) und gibt ihn auf den physischen Ausgang `Output_Q1` aus. Der Wert wird dauerhaft gehalten, bis das Flip-Flop seinen Zustand ändert.

Programmablauf und Verbindungen

1. **Eingangsereignis**: Der Baustein `DigitalInput_CLK_I1` wartet auf einen Tastendruck am Eingang `Input_I1`. Sobald das Ereignis `BUTTON_SINGLE_CLICK` auftritt, wird ein Ereignis am Ausgang `IND` erzeugt.
2. **Ereignisverbindung**: Das Ereignis `IND` wird direkt an den Ereigniseingang `CLK` des Toggle-Flip-Flops `AX_T_FF` weitergeleitet.
3. **Zustandsänderung**: Das Flip-Flop `AX_T_FF` schaltet bei jedem `CLK`-Ereignis seinen internen Zustand um. Der aktuelle Zustand liegt am Adapterausgang `Q` an.
4. **Adapterverbindung**: Der Adapterausgang `Q` des Flip-Flops ist mit dem Adaptereingang `OUT` des Ausgangsbausteins `DigitalOutput_Q1` verbunden. Dadurch wird der neue Zustand sofort an den physischen Ausgang `Output_Q1` übergeben.
5. **Ausgangszustand**: Nach einem Systemstart oder RESET bleibt der Ausgang auf dem initialen Wert `FALSE` (0). Bei jedem weiteren Tastendruck wechselt der Ausgang zwischen `TRUE` (1) und `FALSE` (0).

Hinweise zur praktischen Durchführung:

- Die Übung setzt voraus, dass ein logiBUS-IO-Modul mit einem Taster an `Input_I1` und einer Anzeige (z. B. LED) an `Output_Q1` angeschlossen ist.
- Das Verhalten ist entprellt, da das Ereignis `BUTTON_SINGLE_CLICK` bereits eine gefilterte Flanke liefert.
- Die Bausteine sind so konfiguriert, dass sie automatisch aktiv sind (`QI = TRUE`).

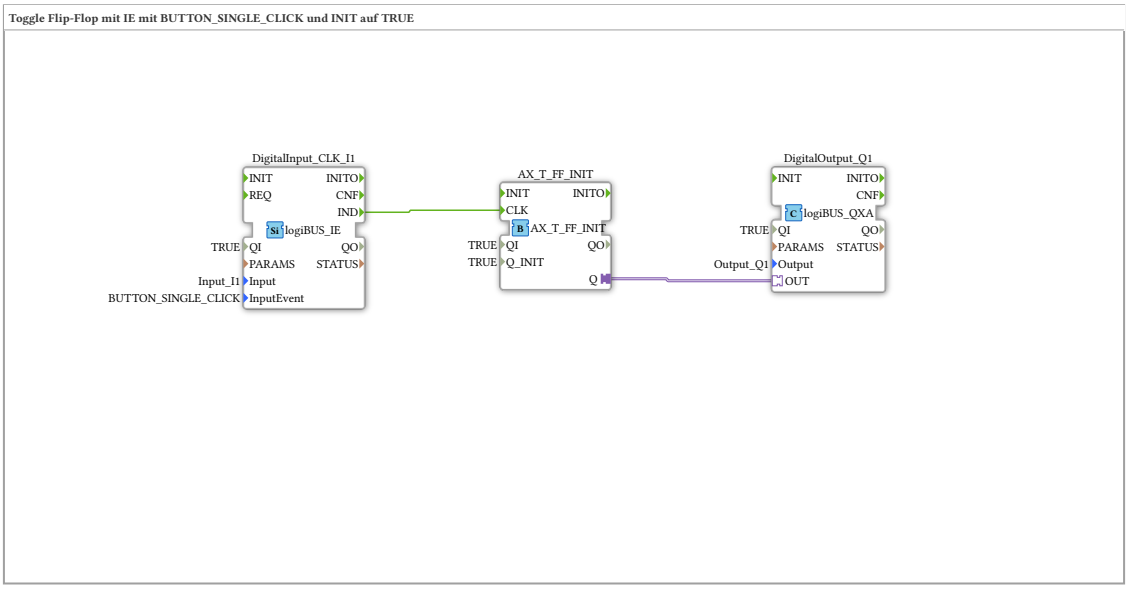
Zusammenfassung

Die Übung demonstriert die einfache Realisierung eines Toggle-Flip-Flops mit hardwarenahen Ein- und Ausgängen. Durch die Kombination eines digitalen Eingangsbausteins mit einem Standard-Flip-Flop und einem Ausgangsbaustein wird ein praktischer Anwendungsfall der Ereignissteuerung in 4diac umgesetzt. Der Schwerpunkt liegt auf dem Verständnis der Ereignisverbindungen und der Initialisierung von Zuständen.

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

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

2.1.2.43 Uebung_004a10b_AX: Toggle Flip-Flop mit IE mit BUTTON_SINGLE_CLICK und INIT auf TRUE



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung realisiert einen asynchronen Toggle-Flip-Flop (T-FF) unter Verwendung eines logiBUS-Eingangsbausteins mit dem Ereignis `BUTTON_SINGLE_CLICK` und einem logiBUS-Ausgangsbaustein. Der Funktionsbaustein wird initial auf `TRUE` gesetzt – der Ausgang ist sofort nach dem Start eingeschaltet. Ein Tastendruck auf den Eingang `I1` löscht den Ausgang; ein weiterer Tastendruck schaltet ihn wieder ein (Toggle-Verhalten).

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_CLK_I1** – Typ: `logiBUS::io::DI::logiBUS_IE`
  - Parameter:
    - `QI` = `TRUE` (Baustein aktiv)
    - `Input` = `Input_I1` (Hardware-Eingangskanal)
    - `InputEvent` = `BUTTON_SINGLE_CLICK` (Ereignis bei kurzem Tastendruck)
  - Ereignisausgang: `IND` (wird bei einem gültigen Tastendruck ausgelöst)
  - **Funktion**: Erzeugt bei einem einzelnen Tastendruck auf den physischen Eingang `I1` ein Ereignis am Ausgang `IND`.
- **AX_T_FF** – Typ: `adapter::events::unidirectional::AX_T_FF_INIT`
  - Parameter:
    - `QI` = `TRUE` (Baustein aktiv)
    - `Q_INIT` = `TRUE` (Anfangswert des Ausgangs)
  - Ereigniseingang: `CLK` (Taktsignal zum Umschalten)
  - Adapterausgang: `Q` (enthält sowohl das Ereignis als auch den Datenwert)
  - **Funktion**: Implementiert ein Toggle-Flip-Flop (T-FF). Bei jedem Ereignis am `CLK`-Eingang wird der interne Zustand umgeschaltet. Der initiale Zustand wird durch `Q_INIT` festgelegt. Der Ausgang `Q` gibt den aktuellen Zustand als Adapterverbindung weiter.
- **DigitalOutput_Q1** – Typ: `logiBUS::io::DQ::logiBUS_QXA`
  - Parameter:
    - `QI` = `TRUE` (Baustein aktiv)
    - `Output` = `Output_Q1` (Hardware-Ausgangskanal)

- Adaptereingang: `OUT` (nimmt den Zustand über eine Adapterverbindung entgegen)
- **Funktion**: Setzt den physischen Ausgang `Q1` auf den Wert, der über die Adapterverbindung ankommt.

Programmablauf und Verbindungen

1. **Startverhalten**: Der Baustein `AX_T_FF` hat `Q_INIT = TRUE`. Dadurch ist der Ausgang `Q` unmittelbar nach dem Start aktiv. Über die Adapterverbindung `AX_T_FF.Q → DigitalOutput_Q1.OUT` wird der Wert an den Ausgangsbaustein weitergegeben, sodass der physische Ausgang `Q1` sofort eingeschaltet ist – dies wird durch den Kommentar „am Anfang gleich mal AN !“ verdeutlicht.
2. **Toggle-Ablauf**:
   - Wenn die Taste an `I1` einmal kurz gedrückt wird, erzeugt `DigitalInput_CLK_I1` ein Ereignis am Ausgang `IND`.
   - Dieses Ereignis wird über die Ereignisverbindung `DigitalInput_CLK_I1.IND → AX_T_FF.CLK` an den Takt-Eingang des T-FF weitergeleitet.
   - Der T-FF kippt daraufhin seinen internen Zustand: Aus `TRUE` wird `FALSE` (oder umgekehrt).
   - Der neue Zustand wird über die Adapterverbindung an den Ausgangsbaustein übertragen, der daraufhin den physischen Ausgang entsprechend setzt.
3. **Wiederholtes Drücken**: Jeder weitere `BUTTON_SINGLE_CLICK` löst erneut ein Umschalten aus, sodass der Ausgang zwischen `TRUE` und `FALSE` hin- und herschaltet.

Verbindungsübersicht:

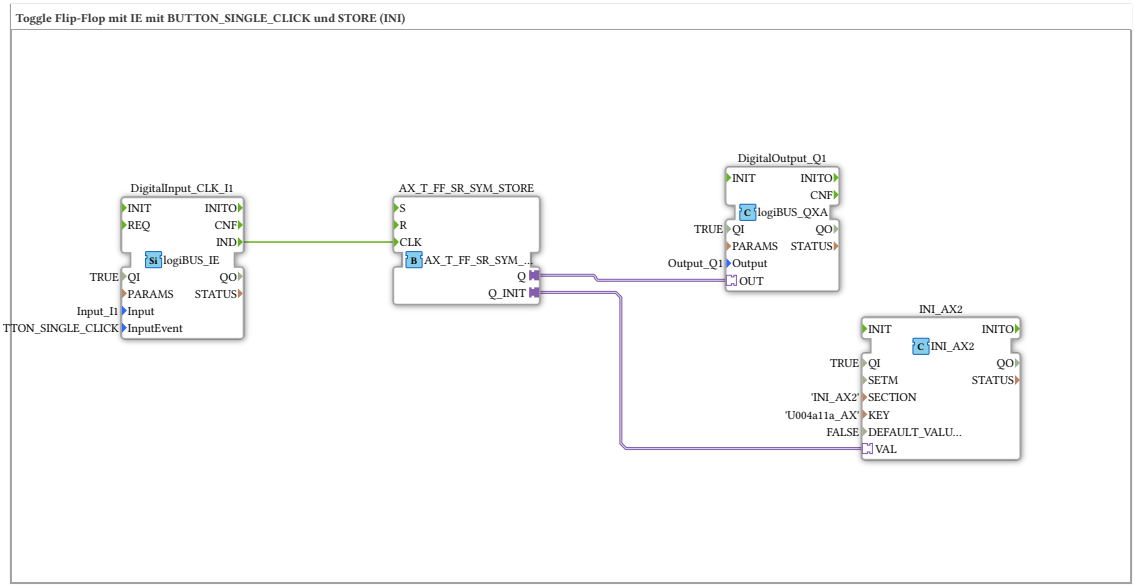
- **Ereignisverbindung**: `DigitalInput_CLK_I1.IND` ⟶ `AX_T_FF.CLK`
- **Adapterverbindung**: `AX_T_FF.Q` ⟶ `DigitalOutput_Q1.OUT`

Zusammenfassung

Die Übung demonstriert den Aufbau eines Toggle-Flip-Flops mit einem initialen Zustand (`TRUE`). Der Eingangsbaustein reagiert ausschließlich auf einen einzelnen Tastendruck (`BUTTON_SINGLE_CLICK`), wodurch Prellen oder Mehrfachauslösungen vermieden werden. Der T-FF wird als vorgefertigter Adapterbaustein (`AX_T_FF_INIT`) verwendet, der sowohl den Umschaltmechanismus als auch die Initialisierung vereint. Die Adapterverbindung zwischen T-FF und Ausgang vereinfacht die Kopplung von Ereignis- und Datenfluss.

Nach dem Start leuchtet die Ausgangslampe `Q1` sofort. Jeder Tastendruck schaltet sie um – ein einfaches und robustes Ein-/Aus-Verhalten.

2.1.2.44 Uebung_004a11a_AX: Toggle Flip-Flop mit IE mit BUTTON_SINGLE_CLICK und STORE (INI)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung realisiert einen Toggle-Flip-Flop (T-Flip-Flop) zur Ansteuerung eines digitalen Ausgangs über einen einzelnen Tastendruck. Der letzte Zustand des Ausgangs wird beim Start der Applikation automatisch aus einem nichtflüchtigen Speicher geladen und beibehalten. Dadurch bleibt der Schaltzustand auch nach einem Neustart erhalten. Die Eingabe erfolgt über einen Taster mit Entprellung, der ein **BUTTON_SINGLE_CLICK**-Ereignis auslöst.

Verwendete Funktionsbausteine (FBs)

Die SubApplikation nutzt folgende vorgefertigte Funktionsbausteine:

- **DigitalInput_CLK_I1** (Typ `logiBUS::io::DI::logiBUS_IE`) Digitaler Eingang, der bei einem einzelnen Tastendruck das Ereignis `IND` erzeugt. Parameter: `QI = TRUE`, `Input = Input_I1`, `InputEvent = BUTTON_SINGLE_CLICK`.
- **AX_T_FF** (Typ `adapter::events::unidirectional::AX_T_FF_SR_SYM_STORE`) Adapterbaustein eines Toggle-Flip-Flops mit symmetrischem SR-Eingang und integrierter Speicherfunktion. Das Flip-Flop wechselt bei jedem steigenden Impuls am Takteingang (`CLK`) seinen Ausgangszustand (`Q`). Über den Adapteranschluss `Q_INIT` wird der initiale Zustand aus einem Speicherbaustein geladen.
- **DigitalOutput_Q1** (Typ `logiBUS::io::DQ::logiBUS_QXA`) Digitaler Ausgang zur Ansteuerung einer Last (z. B. Lampe oder Relais). Parameter: `QI = TRUE`, `Output = Output_Q1`.
- **INI_AX2** (Typ `eclipse4diac::storage::INI_AX2`) Speicherbaustein zum Lesen des gespeicherten Zustands aus einer INI-Datei (oder einem ähnlichen persistenten Medium). Parameter: `QI = TRUE`, `SECTION = 'INI_AX2'`, `KEY = 'U004a11a_AX'`, `DEFAULT_VALUE = FALSE`. Beim Start liefert dieser Baustein den zuletzt gespeicherten Ausgangswert.

Programmablauf und Verbindungen

Die SubApplikation besitzt keine eigenen Ein-/Ausgangsschnittstellen; die Verbindungen sind vollständig intern realisiert. Der Ablauf gliedert sich in zwei Phasen:

1. **Initialisierungsphase (Start)**
   - Der Baustein `INI_AX2` wird aktiviert und liest den unter dem Schlüssel `U004a11a_AX` gespeicherten Wert aus der INI-Datei.
   - Über die Adapterverbindung `AX_T_FF.Q_INIT → INI_AX2.VAL` wird dieser Wert an das Flip-Flop übergeben, das seinen internen Zustand entsprechend setzt.
   - Anschließend gibt das Flip-Flop diesen Zustand über die Adapterverbindung `AX_T_FF.Q → DigitalOutput_Q1.OUT` an den Ausgang weiter.
2. **Betriebsphase (wiederholte Tastendrücke)**
   - Bei jedem einzelnen Tastendruck am Eingang `Input_I1` erzeugt der Baustein `DigitalInput_CLK_I1` das Ereignis `IND`.
   - Dieses Ereignis wird über die Eventverbindung `DigitalInput_CLK_I1.IND → AX_T_FF.CLK` als Taktsignal an das Flip-Flop geleitet.
   - Das Flip-Flop wechselt daraufhin seinen Zustand (von `TRUE` nach `FALSE` oder umgekehrt).
   - Der neue Zustand wird wiederum über die Adapterverbindung an den Ausgang `DigitalOutput_Q1` weitergegeben und gleichzeitig im Speicherbaustein (implizit durch den Adapter) abgelegt.

Ein Kommentar im Netzwerk weist darauf hin, dass der letzte Zustand zu Beginn geladen werden muss.

Zusammenfassung

Die Übung demonstriert die Kombination eines entprellten Tastereingangs mit einem speichernden Toggle-Flip-Flop. Besonders wichtig ist die Wiederherstellung des letzten Ausgangszustands nach einem Neustart – erreicht durch den Einsatz eines INI-Speicherbausteins. Dadurch eignet sich die Schaltung für Anwendungen, bei denen der Schaltzustand auch nach Spannungsunterbrechung erhalten bleiben muss, z. B. für EIN/AUS-Taster in Steuerungen.

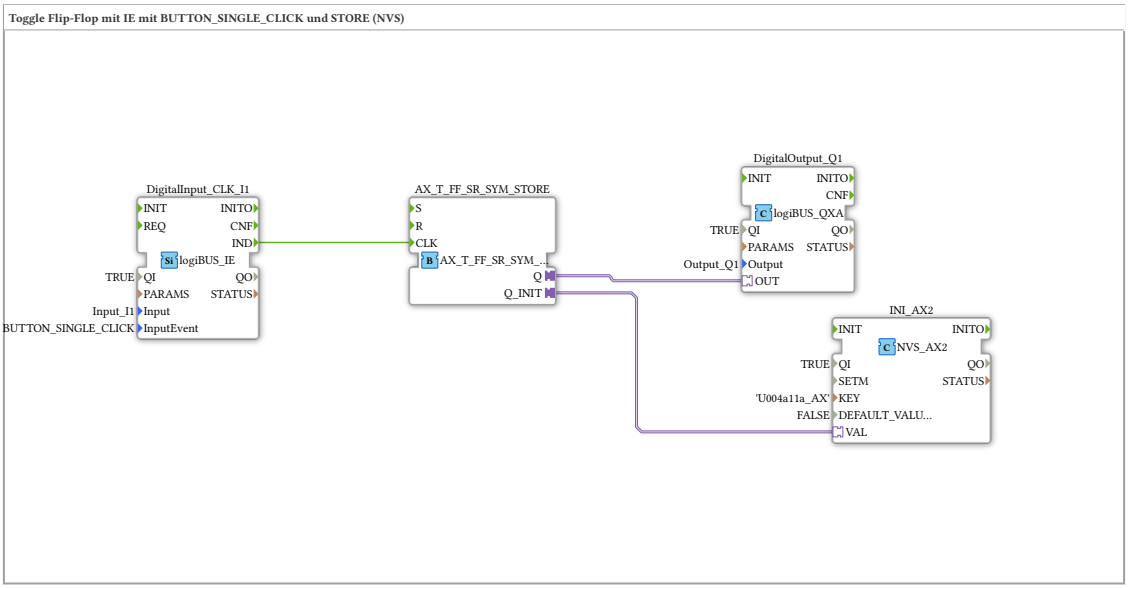
Lernziele:

- Verständnis des Toggle-Flip-Flop-Verhaltens
- Umgang mit Ereignis-gesteuerten Eingängen (BUTTON_SINGLE_CLICK)
- Initialisierung von Zuständen aus persistentem Speicher
- Adapterverbindungen zwischen Funktionsbausteinen in 4diac-IDE

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

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

2.1.2.45 Uebung_004a11b_AX: Toggle Flip-Flop mit IE mit BUTTON_SINGLE_CLICK und STORE (NVS)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung realisiert ein **Toggle-Flip-Flop** (T-FF) mit einem digitalen Eingang (logiBUS DI), der über das Ereignis `BUTTON_SINGLE_CLICK` ausgelöst wird. Der aktuelle Zustand des Flip-Flops wird zudem in einem nichtflüchtigen Speicher (NVS) abgelegt und beim Start wiederhergestellt. Dadurch bleibt der Ausgangszustand auch nach einem Neustart erhalten.

Die Übung demonstriert die Kombination von ereignisgesteuerter Logik (T-FF) mit persistenter Datenspeicherung.

Verwendete Funktionsbausteine (FBs)

Die Übung besteht aus insgesamt vier Funktionsbausteinen, die in einem Subapplikationsnetzwerk verbunden sind. Es gibt keine weiteren eingebetteten Sub-Bausteine.

- **DigitalInput_CLK_I1** (Typ: `logiBUS_IE`)
  - Parameter:
    - `QI` = 1 (aktiv)
    - `Input` = `Input_I1` (physikalischer DI)
    - `InputEvent` = `BUTTON_SINGLE_CLICK`
  - Funktion: Der Baustein erwartet einen Tastendruck auf dem digitalen Eingang. Bei jeder steigenden Flanke (Single Click) wird ein Ereignis am Ausgang `IND` erzeugt.
- **AX_T_FF** (Typ: `AX_T_FF_SR_SYM_STORE`)
  - Parameter: Keine
  - Funktion: Ein T-Flip-Flop mit Set/Reset und Speicherfähigkeit.
    Der Eingang `CLK` (Event) toggelt den internen Zustand bei jedem Ereignis.
    Die Adapterausgänge:
    - `Q` – aktueller Ausgangszustand (an Digitalausgang weitergegeben)
    - `Q_INIT` – dient zur Initialisierung des Flip-Flops mit einem gespeicherten Wert (über NVS)
- **INI_AX2** (Typ: `NVS_AX2`)
  - Parameter:

    - QI = 1 (aktiv)
    - KEY = U004a11a_AX
    - DEFAULT_VALUE = 0 (FALSE)
  - Funktion: Liest beim Start den unter dem Schlüssel U004a11a_AX gespeicherten Wert aus dem NVS (nichtflüchtiger Speicher) aus.
    Der gelesene Wert wird am Adapterausgang VAL bereitgestellt. Falls noch kein Wert gespeichert ist, wird DEFAULT_VALUE (FALSE) ausgegeben.
- **DigitalOutput_Q1** (Typ: logiBUS_QXA)
  - Parameter:
    - QI = 1 (aktiv)
    - Output = Output_Q1 (physikalischer DO)
  - Funktion: Der Baustein gibt den am Adaptereingang OUT anliegenden Wert direkt auf den digitalen Ausgang Q1 aus.

Programmablauf und Verbindungen

Die Verbindungen zwischen den Bausteinen definieren den Ablauf:

1. **Eingangsereignis**
   Bei einem Tastendruck am Eingang I1 erzeugt DigitalInput_CLK_I1 ein Ereignis an seinem Ausgang IND.
2. **Toggle-Flip-Flop**
   Dieses Ereignis wird über eine **Event-Verbindung** direkt an den Eingang CLK des T-FF AX_T_FF weitergeleitet. Jedes Ereignis toggelt den internen Zustand des Flip-Flops.
3. **Ausgang**
   Der aktuelle Ausgangszustand des T-FF (Q) wird über eine **Adapterverbindung** an den Eingang OUT des Digitalausgangs DigitalOutput_Q1 übergeben. Somit wird der toggelte Zustand auf dem physikalischen Ausgang Q1 sichtbar.
4. **Speicherung und Initialisierung**
   Der Baustein INI_AX2 liest beim Start den gespeicherten Wert aus dem NVS und stellt ihn an seinem Adapterausgang VAL bereit.
   Dieser Wert wird über eine **Adapterverbindung** an den Initialisierungseingang Q_INIT des T-FF geführt. Dadurch wird das Flip-Flop beim Start auf den zuletzt gespeicherten Zustand gesetzt (siehe Kommentar im Netzwerk: „am Anfang letzten Zustand laden!“).

   Hinweis: Die Rückspeicherung des aktuellen Zustands in den NVS wird vermutlich intern durch den T-FF oder einen weiteren Baustein realisiert; in der vorliegenden Netzwerkgrafik ist die Speicherung nicht als separater FB sichtbar.

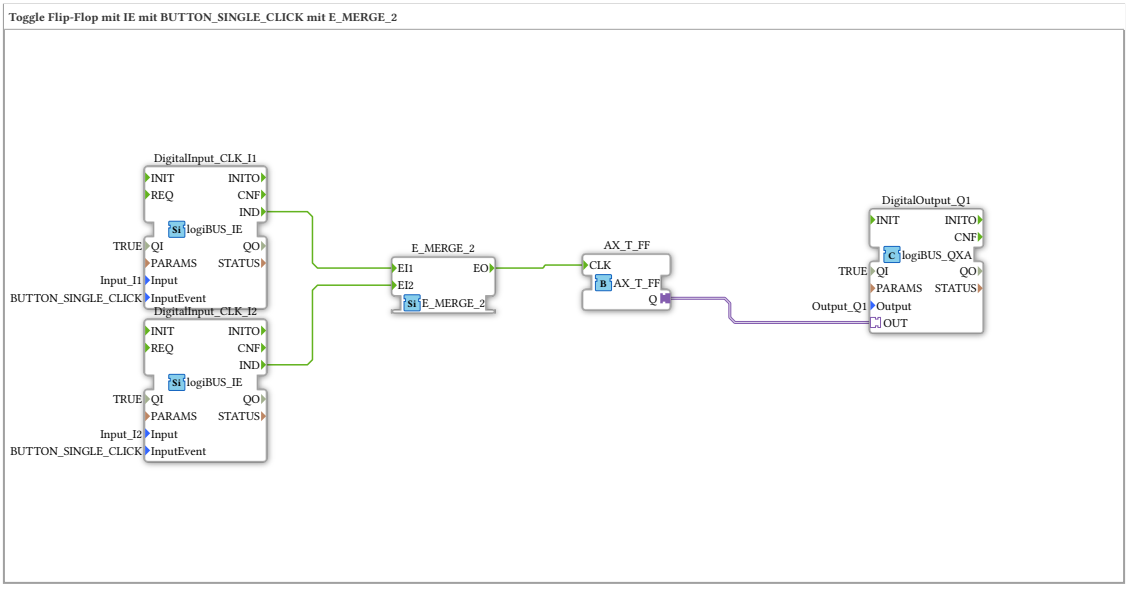
Zusammenfassung

Die Übung veranschaulicht:

- Die Erzeugung von Ereignissen durch einen digitalen Taster mit BUTTON_SINGLE_CLICK.
- Das Toggeln eines Z-Ustands mittels eines T-Flip-Flops.
- Die persistente Speicherung des Zustands in einem nichtflüchtigen Speicher (NVS) und dessen Wiederherstellung nach einem Neustart.
- Der Aufbau einer einfachen ereignisgesteuerten Logik mit kombinierten Ein-/Ausgängen (DI/DO).

Damit sind die Grundlagen für den Einsatz von T-Flip-Flops in Kombination mit NVS-Speicherung für typische Anwendungen wie das Ein- und Ausschalten von Ausgängen mit Zustandserhaltung gelegt.

2.1.2.46 Uebung_004a2_2_AX: Toggle Flip-Flop mit IE mit BUTTON_SINGLE_CLICK mit E_MERGE_2



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung realisiert ein Toggle-Flipflop (T-FF), das durch zwei unabhängige Taster (Eingänge I1 und I2) umgeschaltet wird. Jeder Taster löst ein Ereignis „BUTTON_SINGLE_CLICK“ aus. Die beiden Ereignisse werden mithilfe eines `E_MERGE_2`-Bausteins zusammengeführt und dienen als Taktsignal für das T-FF. Der Ausgang des T-FF steuert einen digitalen Ausgang (Q1).

Verwendete Funktionsbausteine (FBs)

- **DigitalOutput_Q1**
  - **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
  - **Parameter**:
    - `QI` = TRUE
    - `PARAMS` = ““
    - `Output` = `Output_Q1`
  - **Beschreibung**: Stellt den digitalen Ausgang Q1 bereit. Der Ausgangswert wird über den Adaptereingang gesetzt.
- **DigitalInput_CLK_I1**
  - **Typ**: `logiBUS::io::DI::logiBUS_IE`
  - **Parameter**:
    - `QI` = TRUE
    - `PARAMS` = ““
    - `Input` = `Input_I1`
    - `InputEvent` = `BUTTON_SINGLE_CLICK`
  - **Beschreibung**: Liest den digitalen Eingang I1 und erzeugt bei einem kurzen Tastendruck (Single Click) ein Ereignis `IND`.
- **DigitalInput_CLK_I2**
  - **Typ**: `logiBUS::io::DI::logiBUS_IE`
  - **Parameter**:
    - `QI` = TRUE
    - `PARAMS` = ““
    - `Input` = `Input_I2`
    - `InputEvent` = `BUTTON_SINGLE_CLICK`

  - **Beschreibung**: Liest den digitalen Eingang I2 und erzeugt bei einem kurzen Tastendruck ein Ereignis `IND`.

- **E_MERGE_2**
  - **Typ**: `iec61499::events::E_MERGE_2`
  - **Parameter**: Keine
  - **Beschreibung**: Vereinigt zwei Ereigniseingänge (EI1, EI2) zu einem gemeinsamen Ereignisausgang (EO). Sobald an einem der beiden Eingänge ein Ereignis anliegt, wird es an den Ausgang weitergegeben.

- **AX_T_FF**
  - **Typ**: `adapter::events::unidirectional::AX_T_FF`
  - **Parameter**: Keine
  - **Beschreibung**: Ein Toggle-Flipflop als Adapter. Bei jedem Ereignis am Eingang `CLK` wechselt der Ausgang `Q` seinen Zustand (0⟶1, 1⟶0).

Programmablauf und Verbindungen

Das System arbeitet ereignisgesteuert. Sobald der Benutzer den Taster an Eingang I1 oder I2 kurz drückt, erzeugt der entsprechende `DigitalInput`-Baustein ein Ereignis `IND`. Diese beiden Ereignisse werden über den `E_MERGE_2`-Baustein zusammengeführt – unabhängig davon, welcher Taster gedrückt wurde, löst das `E_MERGE_2` ein Ereignis an seinem Ausgang `EO` aus. Dieses Ereignis wird direkt an den Takteingang `CLK` des T-FF (`AX_T_FF`) weitergeleitet. Das T-FF toggelt seinen Ausgangszustand bei jedem eingehenden Ereignis. Der aktuelle Zustand `Q` des T-FF wird über eine Adapterverbindung an den digitalen Ausgang `DigitalOutput_Q1` übergeben, der den physischen Ausgang Q1 ansteuert.

Verbindungsübersicht:

- Ereignisverbindungen:
  - `DigitalInput_CLK_I1.IND` ⟶ `E_MERGE_2.EI1`
  - `DigitalInput_CLK_I2.IND` ⟶ `E_MERGE_2.EI2`
  - `E_MERGE_2.EO` ⟶ `AX_T_FF.CLK`
- Adapterverbindungen:
  - `AX_T_FF.Q` ⟶ `DigitalOutput_Q1.OUT`

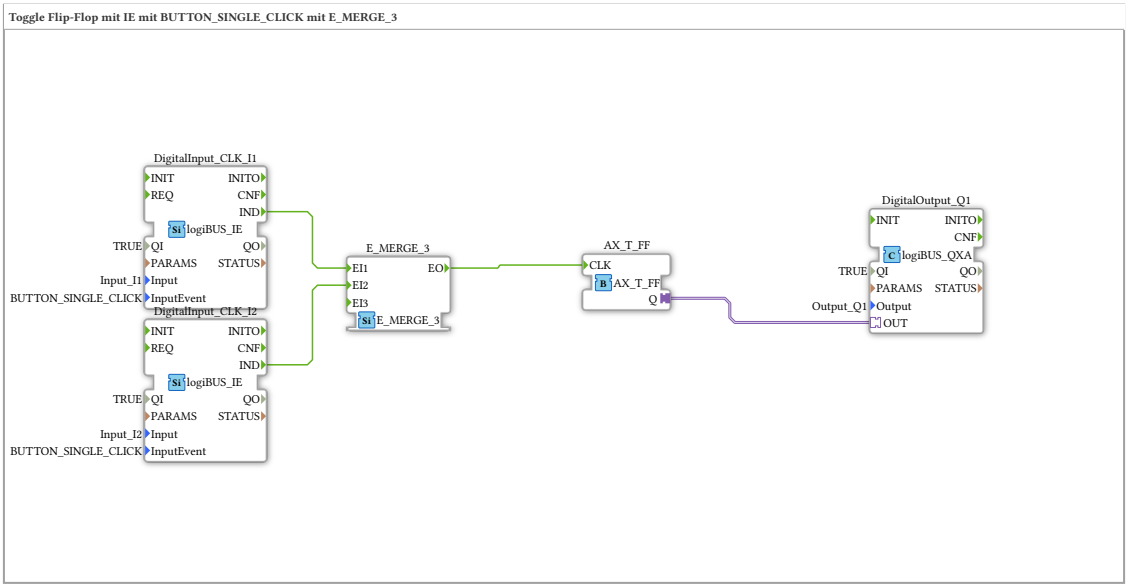
Zusammenfassung

Die Übung demonstriert die Kombination von zwei Ereignisquellen (Taster) mit einem `E_MERGE_2`-Baustein zur Erzeugung eines gemeinsamen Taktsignals für ein Toggle-Flipflop. Der Ausgangszustand des Flipflops kann durch beliebiges Drücken der beiden Taster umgeschaltet werden. Dies ist ein grundlegendes Beispiel für die ereignisgesteuerte Logik in 4diac und die Verwendung adaptierbarer I/O-Bausteine aus der logiBUS-Bibliothek.

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

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

2.1.2.47 Uebung_004a2_3_AX: Toggle Flip-Flop mit IE mit BUTTON_SINGLE_CLICK mit E_MERGE_3



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung realisiert ein **Toggle Flip-Flop** (T-FF), das über zwei separate Taster (Eingänge I1 und I2) gesteuert wird. Die Taster werden als **BUTTON_SINGLE_CLICK** konfiguriert, d.h. jeder Tastendruck erzeugt genau ein Ereignis. Die Ereignisse beider Taster werden über einen **E_MERGE_3** Baustein zusammengeführt und an den Takteingang (CLK) des T-FF weitergeleitet. Der Ausgang Q des T-FF schaltet einen digitalen Ausgang (Q1). Das Schaltverhalten: Jeder Tastendruck (egal von welchem Taster) toggelt den Ausgangszustand.

Die Übung demonstriert die Kombination von Hardware-Eingangsbausteinen mit Ereignisverarbeitung und einem Adapter-basierten Flip-Flop.

Verwendete Funktionsbausteine (FBs)

Im Netzwerk werden folgende Funktionsbausteine eingesetzt:

- **DigitalOutput_Q1** – Typ: `logiBUS::io::DQ::logiBUS_QXA`
  - Parameter: `QI` = `TRUE`, `Output` = `Output_Q1`
  - Aufgabe: Stellt den digitalen Ausgang Q1 bereit. Der Ausgangswert wird über den Adaptereingang `OUT` gesetzt.
- **DigitalInput_CLK_I1** – Typ: `logiBUS::io::DI::logiBUS_IE`
  - Parameter: `QI` = `TRUE`, `Input` = `Input_I1`, `InputEvent` = `BUTTON_SINGLE_CLICK`
  - Aufgabe: Erfasst den Taster an I1. Bei jedem Tastendruck wird das Ereignis `BUTTON_SINGLE_CLICK` ausgelöst und ein IND-Ereignis am Ausgang gesendet.
- **DigitalInput_CLK_I2** – Typ: `logiBUS::io::DI::logiBUS_IE`
  - Parameter: `QI` = `TRUE`, `Input` = `Input_I2`, `InputEvent` = `BUTTON_SINGLE_CLICK`
  - Aufgabe: Erfasst den Taster an I2. Gleiche Funktionsweise wie der Baustein für I1.
- **AX_T_FF** – Typ: `adapter::events::unidirectional::AX_T_FF`
  - Parameter: keine zusätzlichen
  - Aufgabe: Adapterbaustein, der ein T-Flip-Flop realisiert. Bei jedem Ereignis am CLK-Eingang wird der interne Zustand umgeschaltet. Der Ausgang Q (Adapter) gibt den aktuellen Zustand aus.
- **E_MERGE_3** – Typ: `iec61499::events::E_MERGE_3`
  - Parameter: keine zusätzlichen
  - Aufgabe: Ereigniszusammenführung. Die drei Ereigniseingänge (EI1, EI2, EI3) werden logisch ODER-verknüpft; bei jedem eingehenden Ereignis an einem der Eingänge wird ein Ereignis am Ausgang EO ausgegeben. In dieser Übung werden nur zwei Eingänge (EI1, EI2) verwendet; der dritte bleibt unbeschaltet.

Programmablauf und Verbindungen

Die Verdrahtung arbeitet wie folgt:

1. **Eingangsereignisse**: Die beiden DigitalInput-Bausteine erzeugen bei Betätigung des jeweiligen Tasters ein IND-Ereignis (Tastendruck erkannt).
2. **Ereignis-Merge**: Die IND-Ereignisse werden zu `E_MERGE_3` geführt:
   - `DigitalInput_CLK_I1.IND` ⟶ `E_MERGE_3.EI1`
   - `DigitalInput_CLK_I2.IND` ⟶ `E_MERGE_3.EI2`
   - Der dritte Eingang (EI3) ist nicht verbunden (laut Kommentar ist dies zulässig).
3. **Takt für das Flip-Flop**: Das zusammengeführte Ereignis (`E_MERGE_3.EO`) ist mit dem CLK-Eingang des T-FF (`AX_T_FF.CLK`) verbunden. Jeder Tastendruck löst somit ein Taktereignis aus.
4. **Ausgang**: Der Adapterausgang `AX_T_FF.Q` ist mit dem Eingang `OUT` des DigitalOutput-Bausteins verbunden. Der Zustand des Flip-Flops wird direkt auf den digitalen Ausgang Q1 gegeben.

Verhalten: Bei jedem Tastendruck (I1 oder I2) wechselt Q1 seinen Zustand (von 0 ⟶ 1 oder 1 ⟶ 0). Das entspricht einem typischen Toggle-Flip-Flop.

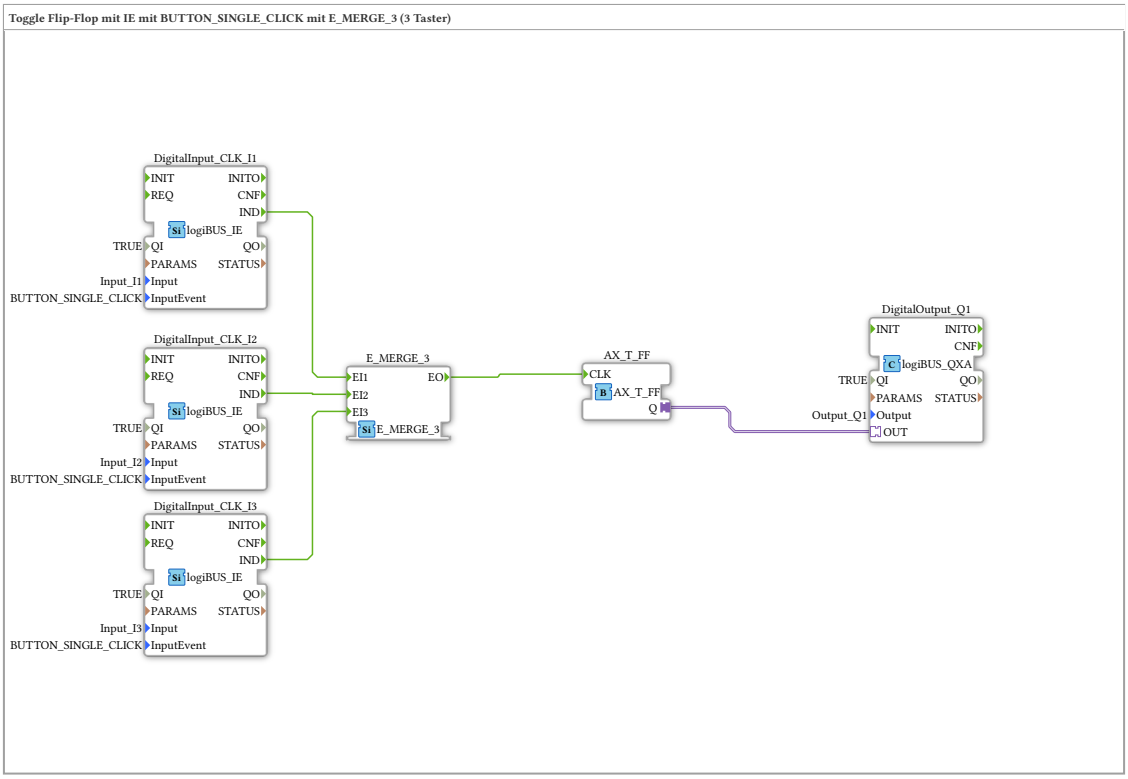
Zusammenfassung

Die Übung zeigt, wie ein **Toggle Flip-Flop** mit Hilfe eines Adapterbausteins (`AX_T_FF`) und Ereignisverarbeitung aufgebaut wird. Zwei Taster werden über `BUTTON_SINGLE_CLICK` konfiguriert und ihre Ereignisse mit einem `E_MERGE_3` zusammengeführt, sodass jeder Tastendruck unabhängig vom anderen das Flip-Flop toggelt. Dies ist eine grundlegende Schaltung zur Ereignissteuerung und Zustandsspeicherung mit 4diac und logiBUS-Hardware.

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

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

2.1.2.48 Uebung_004a2_3b_AX: Toggle Flip-Flop mit IE mit BUTTON_SINGLE_CLICK mit E_MERGE_3 (3 Taster)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung realisiert ein Toggle-Flip-Flop, das über drei Taster (BUTTON_SINGLE_CLICK) gesteuert wird. Die Ereignisse der Taster werden durch einen E_MERGE_3 zu einem einzigen Taktsignal zusammengeführt. Der Flip-Flop wechselt bei jedem Tastendruck seinen Ausgangszustand und gibt das Ergebnis auf einen digitalen Ausgang aus.

Verwendete Funktionsbausteine (FBs)

Sub-Baustein: DigitalInput_CLK_I1 (Typ: `logiBUS_IE`)

- **Typ**: `logiBUS::io::DI::logiBUS_IE`
- **Parameter**:
  - `QI` = `TRUE`
  - `Input` = `Input_I1`
  - `InputEvent` = `BUTTON_SINGLE_CLICK`
- **Funktionsweise**: Digitaler Eingang, der bei einem einzelnen Tastendruck auf Kanal I1 ein Ereignis an seinem Ausgang `IND` auslöst.

Sub-Baustein: DigitalInput_CLK_I2 (Typ: `logiBUS_IE`)

- **Typ**: `logiBUS::io::DI::logiBUS_IE`
- **Parameter**:
  - `QI` = `TRUE`
  - `Input` = `Input_I2`
  - `InputEvent` = `BUTTON_SINGLE_CLICK`
- **Funktionsweise**: Gleicher Baustein wie für Kanal I1, jedoch angeschlossen an den zweiten Taster (Input_I2).

Sub-Baustein: DigitalInput_CLK_I3 (Typ: `logiBUS_IE`)

- **Typ**: `logiBUS::io::DI::logiBUS_IE`
- **Parameter**:
  - `QI` = `TRUE`
  - `Input` = `Input_I3`
  - `InputEvent` = `BUTTON_SINGLE_CLICK`
- **Funktionsweise**: Gleicher Baustein für den dritten Taster (Input_I3).

Sub-Baustein: E_MERGE_3 (Typ: `E_MERGE_3`)

- **Typ**: `iec61499::events::E_MERGE_3`
- **Parameter**: keine
- **Funktionsweise**: Vereinigt drei Ereigniseingänge (`EI1`, `EI2`, `EI3`) zu einem einzigen Ereignisausgang (`EO`). Sobald ein Ereignis an einem der Eingänge eintrifft, wird es unverzögert an `EO` weitergegeben.

Sub-Baustein: AX_T_FF (Typ: `AX_T_FF`)

- **Typ**: `adapter::events::unidirectional::AX_T_FF`
- **Parameter**: keine
- **Funktionsweise**: Toggle-Flip-Flop. Bei jedem Ereignis am Eingang `CLK` wechselt der Zustand am Ausgang `Q` zwischen `TRUE` und `FALSE` (toggle).

Sub-Baustein: DigitalOutput_Q1 (Typ: `logiBUS_QXA`)

- **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
- **Parameter**:
  - `QI` = `TRUE`
  - `Output` = `Output_Q1`
- **Funktionsweise**: Digitaler Ausgang. Der über den Adaptereingang `OUT` empfangene Wert wird auf den logiBUS-Kanal Q1 ausgegeben.

Programmablauf und Verbindungen

Die drei Taster (I1, I2, I3) sind jeweils an einen `logiBUS_IE` angeschlossen, der bei jedem Tastendruck ein Ereignis (`BUTTON_SINGLE_CLICK`) erzeugt. Die Ereignisausgänge dieser drei Eingänge (`DigitalInput_CLK_I1.IND`, `DigitalInput_CLK_I2.IND`, `DigitalInput_CLK_I3.IND`) sind mit den drei Eingängen des `E_MERGE_3` (`EI1`, `EI2`, `EI3`) verbunden. Der Merge-Baustein leitet jedes eingehende Ereignis

an seinen Ausgang `EO` weiter. Dieser ist wiederum mit dem Takteingang `CLK` des Toggle-Flip-Flops `AX_T_FF` verbunden.

Der Flip-Flop wechselt bei jedem empfangenen Ereignis seinen Ausgangszustand. Der aktuelle Zustand wird über den Adapterausgang `Q` an den Adaptereingang `OUT` des Ausgangsbausteins `DigitalOutput_Q1` übergeben und erscheint auf dem digitalen Ausgang `Q1`.

Die gesamte Schaltung realisiert somit ein **Toggle-Flip-Flop mit drei Tastern**: Jeder Tastendruck – unabhängig von welchem Taster – toggelt den Ausgang.

Lernziele

- Verständnis des Toggle-Flip-Flop-Verhaltens
- Anwendung des Ereignis-Merge-Bausteins zur Bündelung mehrerer Ereignisquellen
- Einbindung von logiBUS-Ein- und Ausgängen mit Ereignisauslösung
- Aufbau einer einfachen ereignisgesteuerten Schaltung in 4diac-IDE

Schwierigkeitsgrad

Einfach – geeignet für Einsteiger in die IEC 61499-Modellierung mit 4diac-IDE.

Benötigte Vorkenntnisse

- Grundlegendes Verständnis der IEC 61499-Ereignis- und Datenflüsse
- Grundlagen der logiBUS-Konfiguration (Ein-/Ausgangskanäle)

Zusammenfassung

Die Übung `Uebung_004a2_3b_AX` demonstriert den Aufbau eines Toggle-Flip-Flops, das über drei Taster gesteuert wird. Die Tasterereignisse werden mittels `E_MERGE_3` zu einem einzigen Taktsignal zusammengeführt und toggeln den Zustand eines Flip-Flops, der über einen digitalen Ausgang ausgegeben wird. Dieses einfache Beispiel vermittelt grundlegende Konzepte der ereignisgesteuerten Programmierung mit Funktionsbausteinen unter Verwendung der logiBUS-Hardware.

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

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

2.1.2.49 Uebung_004a2_AX: Toggle Flip-Flop mit IE mit BUTTON_SINGLE_CLICK mit E_MERGE



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_004a2_AX`. Hier wird die Stromstoßschaltung erweitert, sodass sie von zwei verschiedenen Tastern aus bedient werden kann. Dazu werden die Ereignisse der beiden Taster zusammengeführt.

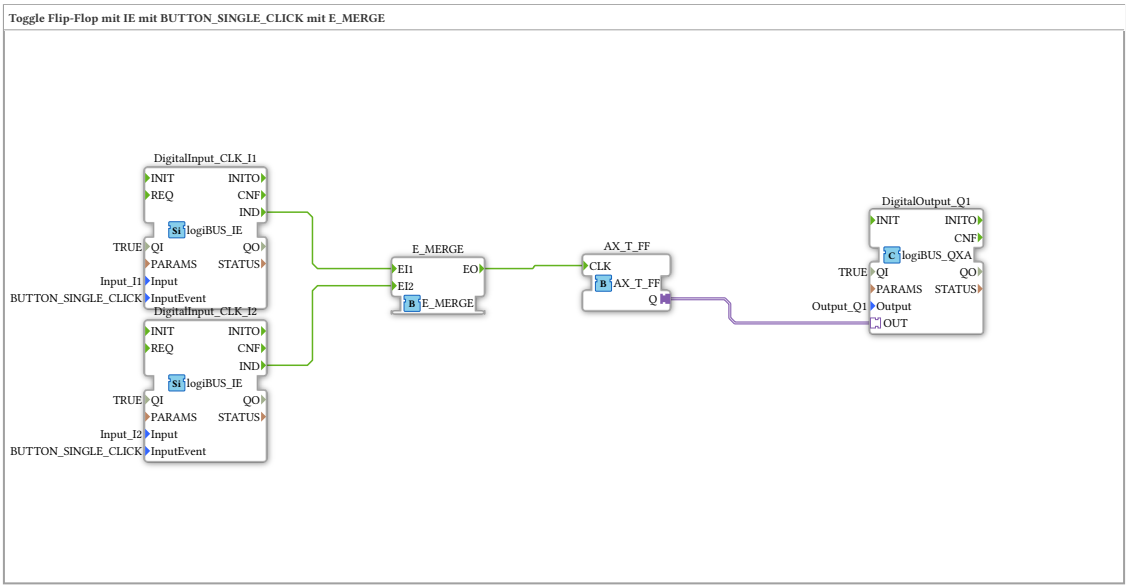
Ziel der Übung

Das Ziel ist es zu lernen, wie man asynchrone Ereignisströme vereint. Wenn zwei Ereignisquellen (Taster) denselben Prozess (Licht umschalten) auslösen sollen, müssen ihre Signale „gemerged“ (zusammengeführt) werden, bevor sie den Funktionsbaustein erreichen.

Beschreibung und Komponenten

Die Subapplikation `Uebung_004a2_AX.SUB` nutzt einen `E_MERGE` Baustein, um zwei Eingangs-Events auf einen Flip-Flop-Eingang zu leiten.

Funktionsbausteine (FBs)



- **`DigitalInput_CLK_I1 & I2`**: Zwei `logiBUS_IE` Bausteine, konfiguriert auf `BUTTON_SINGLE_CLICK`. Sie erzeugen Ereignisse bei Betätigung von Taster 1 oder 2.
- **`E_MERGE`**: Typ `E_MERGE`. Dieser Baustein besitzt zwei Ereigniseingänge (`EI1`, `EI2`) und einen Ereignisausgang (`EO`). Egal welcher Eingang ein Event empfängt, es wird sofort an den Ausgang weitergeleitet.
- **`E_T_FF`**: Das Toggle-Flip-Flop, das den Zustand speichert.
- **`DigitalOutput_Q1`**: Der Ausgang für die Lampe.

Funktionsweise

```
<EventConnections>
    <Connection Source="DigitalInput_CLK_I1.IND" Destination="E_MERGE.EI1"/>
    <Connection Source="DigitalInput_CLK_I2.IND" Destination="E_MERGE.EI2"/>
    <Connection Source="E_MERGE.EO" Destination="E_T_FF.CLK"/>
</EventConnections>
```

1. Drückt man Taster 1, sendet `I1` ein Event an `E_MERGE.EI1`. `E_MERGE` leitet es an `EO` weiter -> `E_T_FF` schaltet um.
2. Drückt man Taster 2, sendet `I2` ein Event an `E_MERGE.EI2`. `E_MERGE` leitet es an `EO` weiter -> `E_T_FF` schaltet um.

Somit kann das Licht von beiden Tastern aus beliebig ein- und ausgeschaltet werden.

Anwendungsbeispiel

Dies entspricht einer **Wechselschaltung im Flur**: Man kann das Licht unten einschalten und oben wieder ausschalten (und umgekehrt). Jeder Taster bewirkt lediglich eine Zustandsänderung („Toggle“), egal wie der aktuelle Zustand ist.

2.1.2.50 Uebung_004a3_AX: wie Uebung_004a2 aber ohne E_MERGE

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_004a3_AX`. Diese Übung zeigt eine Vereinfachung gegenüber `Uebung_004a2_AX`: In IEC 61499 (und speziell in 4diac) können mehrere Ereignisquellen oft direkt auf denselben Ereigniseingang verbunden werden.

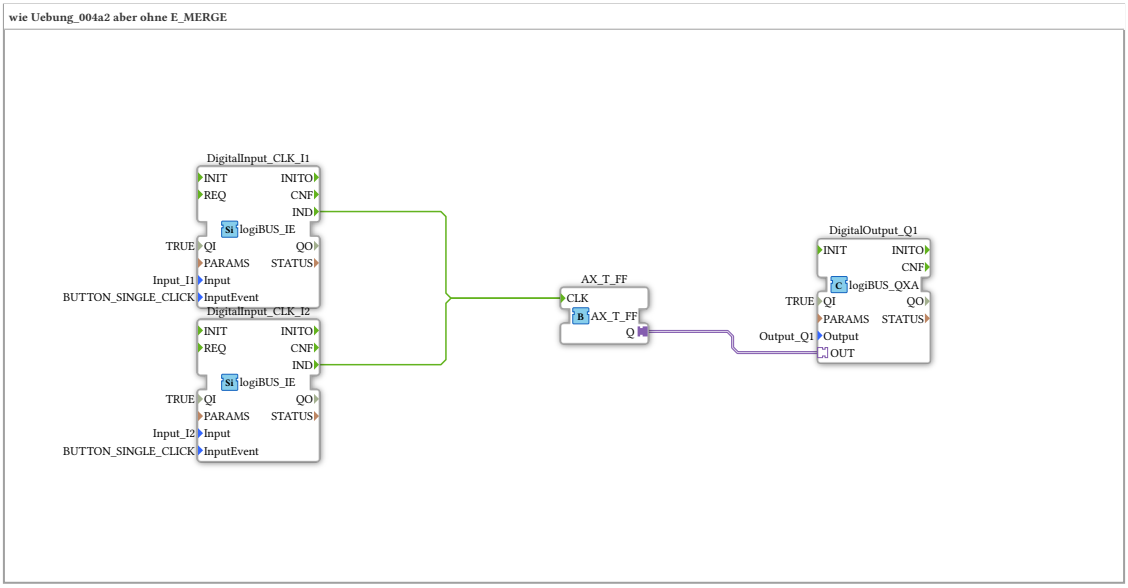
Ziel der Übung

Das Ziel ist die Reduktion der visuellen Komplexität. Es wird demonstriert, dass der explizite `E_MERGE` Baustein in vielen Fällen weggelassen werden kann, da die Laufzeitumgebung (Runtime) eingehende Events an einem Port automatisch nacheinander verarbeitet (implizites ODER für Events).

Beschreibung und Komponenten

Die Subapplikation `Uebung_004a3_AX.SUB` verbindet zwei Event-Quellen direkt mit dem Takteingang des Flip-Flops.

Funktionsbausteine (FBs)



- **`DigitalInput_CLK_I1` & `I2`**: Die Event-Generatoren.
- **`E_T_FF`**: Das Toggle-Flip-Flop.
- **`DigitalOutput_Q1`**: Der Ausgang.

Der Baustein `E_MERGE` fehlt hier bewusst.

Funktionsweise

```
<EventConnections>
    <Connection Source="DigitalInput_CLK_I1.IND" Destination="E_T_FF.CLK"/>
    <Connection Source="DigitalInput_CLK_I2.IND" Destination="E_T_FF.CLK"/>
</EventConnections>
```

Die Funktionsweise ist identisch zur Übung mit `E_MERGE`: Jedes eintreffende Event an `E_T_FF.CLK` – egal ob von `I1` oder `I2` kommend – triggert die Ausführung des Funktionsbausteins. Die 4diac IDE und Runtime unterstützen diese „Fan-In"-Verbindungen für Events.

*(Hinweis: Bei Datenverbindungen ist dies **nicht** erlaubt! Zwei Datenausgänge dürfen niemals direkt auf denselben Dateneingang schreiben, da dies zu Konflikten führen würde. Bei Events ist es jedoch eine gängige Praxis für „ODER"-Verknüpfungen von Auslösern.)*

Anwendungsbeispiel

Gleiches Beispiel wie zuvor (Wechselschaltung), jedoch mit effizienterem Code (weniger Bausteine, weniger Speicherbedarf).

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

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

2.1.2.51 Uebung_004a4_AX: Toggle Flip-Flop mit IE mit BUTTON_SINGLE_CLICK mit E_SPLIT

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_004a4_AX`. Hier wird gezeigt, wie ein einzelnes Ereignis genutzt werden kann, um mehrere unabhängige Prozesse anzustoßen, indem man einen `E_SPLIT` Baustein verwendet.

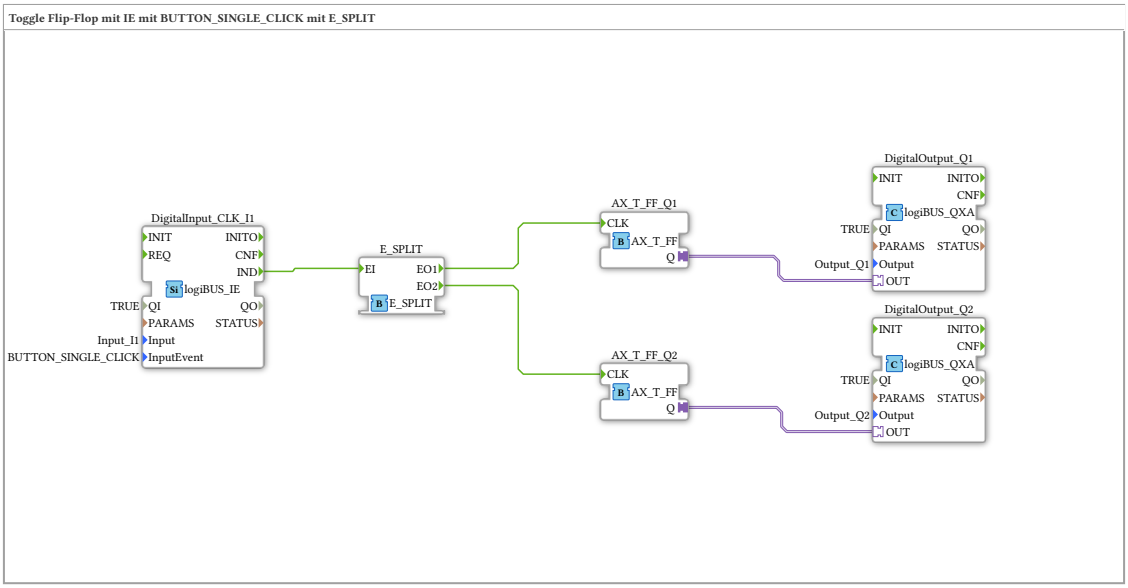
Ziel der Übung

Das Ziel ist das Verständnis der sequenziellen Event-Verarbeitung. In IEC 61499 kann ein Event-Ausgang oft nur mit einem Event-Eingang verbunden sein (Fan-Out = 1), oder man möchte explizit die Reihenfolge der Abarbeitung steuern. Der `E_SPLIT` Baustein nimmt ein Eingangs-Event und feuert nacheinander Ausgänge ab.

Beschreibung und Komponenten

Die Subapplikation `Uebung_004a4_AX.SUB` verwendet einen Taster, um zwei separate Toggle-Flip-Flops zu schalten.

Funktionsbausteine (FBs)



- **`DigitalInput_CLK_I1`**: Der Event-Generator (Taster).
- **`E_SPLIT`**: Ein Event-Verteiler. Er hat einen Eingang `EI` und zwei Ausgänge `EO1` und `EO2`.
- **`E_T_FF_Q1` & `Q2`**: Zwei unabhängige Flip-Flops.
- **`DigitalOutput_Q1` & `Q2`**: Zwei Lampen.

Funktionsweise

```
<EventConnections>
    <Connection Source="DigitalInput_CLK_I1.IND" Destination="E_SPLIT.EI"/>
    <Connection Source="E_SPLIT.EO1" Destination="E_T_FF_Q1.CLK"/>
    <Connection Source="E_SPLIT.EO2" Destination="E_T_FF_Q2.CLK"/>
</EventConnections>
```

1. Ein Klick auf Taster 1 sendet ein Event an `E_SPLIT`.
2. `E_SPLIT` sendet **zuerst** ein Event an `EO1` -> `E_T_FF_Q1` schaltet um.
3. Danach (quasi zeitgleich, aber logisch danach) sendet `E_SPLIT` ein Event an `EO2` -> `E_T_FF_Q2` schaltet um.

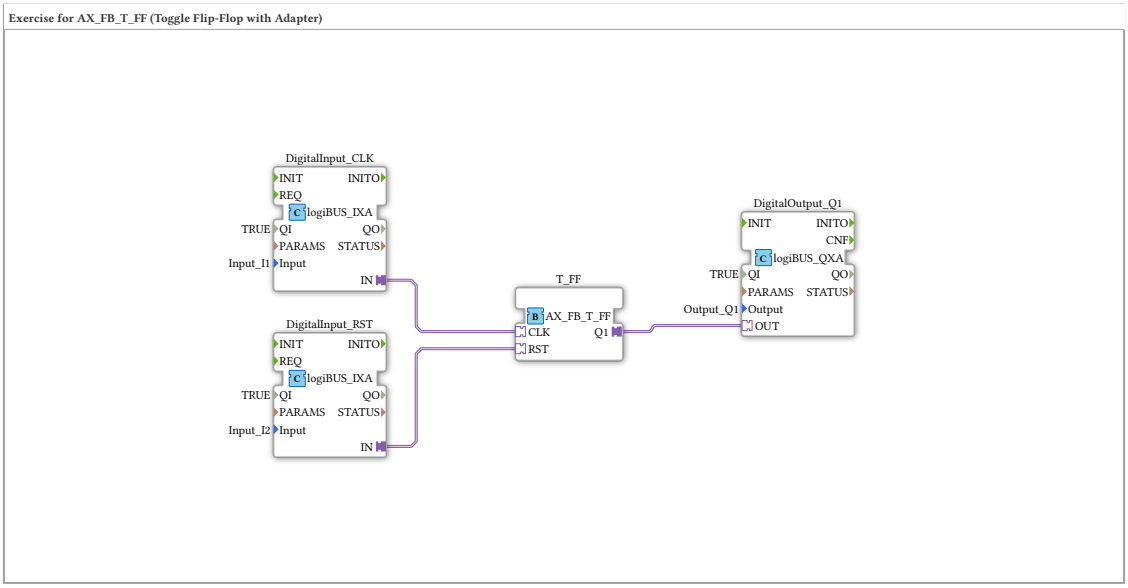
Beide Lampen schalten somit synchron um, gesteuert durch einen Taster.

(Hinweis im Code: „hier 2x T_FF zu verwenden ist sinnlos, das soll nur zeigen wie man E_SPLIT verwenden kann.“ - Das stimmt, man hätte auch beide Ausgänge an ein FF hängen können. Hier geht es rein um die Demonstration des Event-Splittings.)

Anwendungsbeispiel

Szenen-Steuerung: Ein Taster „Feierabend“ betätigt gleichzeitig (bzw. nacheinander) mehrere Aktionen: Licht ausschalten (`Q1`) und Alarmanlage scharfschalten (`Q2`). Durch den Splitter wird sichergestellt, dass beide Funktionsketten angestoßen werden.

2.1.2.52 Uebung_004a4_AX_T: Exercise for AX_FB_T_FF (Toggle Flip-Flop with Adapter)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert die Verwendung eines **Toggle-Flipflops (AX_FB_T_FF)** mit Hilfe von Adaptern. Ein Toggle-Flipflop ändert bei jedem positiven Taktimpuls (CLK) seinen Ausgangszustand (Q1) und kann über den Rücksetzeingang (RST) auf `FALSE` gesetzt werden.

Die Eingänge werden über zwei logiBUS-Digital-Eingabebausteine (Input_I2 für RST und Input_I1 für CLK) realisiert, der Ausgang über einen logiBUS-Digital-Ausgabebaustein (Output_Q1).

Verwendete Funktionsbausteine (FBs)

Die SubApp enthält folgende Funktionsbausteine:

- **DigitalInput_RST**
  - Typ: `logiBUS::io::DI::logiBUS_IXA`
  - Parameter: `QI = TRUE`, `Input = Input_I2`
  - *Verbindet den physischen Eingang Input_I2 mit dem RST-Signal des Flipflops.*
- **DigitalInput_CLK**
  - Typ: `logiBUS::io::DI::logiBUS_IXA`
  - Parameter: `QI = TRUE`, `Input = Input_I1`
  - *Verbindet den physischen Eingang Input_I1 mit dem Taktsignal (CLK) des Flipflops.*
- **T_FF**
  - Typ: `adapter::bistableElements::AX_FB_T_FF`
  - Keine weiteren Parameter.
  - *Kern der Übung: Ein Toggle-Flipflop, das bei jedem Takt den Ausgang umschaltet.*
- **DigitalOutput_Q1**
  - Typ: `logiBUS::io::DQ::logiBUS_QXA`

- Parameter: `QI = TRUE`, `Output = Output_Q1`
- *Verbindet den Flipflop-Ausgang mit dem physischen Ausgang Output_Q1.*

Sub-Bausteine

Es wurden keine weiteren SubApp-Bausteine innerhalb dieser Übung verwendet.

Programmablauf und Verbindungen

Die Funktionsbausteine sind über **Adapterverbindungen** (keine klassischen Event-/Datenverbindungen) verknüpft:

1. **Reset-Signal (RST):** `DigitalInput_RST.IN` ⟶ `T_FF.RST` *Ein Signal auf Eingang Input_I2 setzt das Flipflop zurück (Q1 = FALSE).*
2. **Taktsignal (CLK):** `DigitalInput_CLK.IN` ⟶ `T_FF.CLK` *Jede steigende Flanke auf Input_I1 schaltet den Ausgang Q1 um (von TRUE nach FALSE oder umgekehrt).*
3. **Ausgang (Q1):** `T_FF.Q1` ⟶ `DigitalOutput_Q1.OUT` *Der aktuelle Zustand des Flipflops wird auf den physischen Ausgang Output_Q1 ausgegeben.*

Ablauf:

- Solange kein Reset anliegt, wechselt der Ausgang bei jedem Taktimpuls seinen Zustand.
- Ein aktiver Reset (TRue) setzt den Ausgang sofort auf `FALSE` und hält ihn dort, bis der Reset wieder wegfällt und ein neuer Takt kommt.

Zusammenfassung

Mit dieser Übung wird die Anwendung eines **Toggle-Flipflops** über Adapterverbindungen in der 4diac-IDE vermittelt.

- Lernziele:
  - Verstehen der Arbeitsweise eines Toggle-Flipflops (AX_FB_T_FF).
  - Anbinden von physikalischen Ein-/Ausgängen über logiBUS-Adapter.
  - Erstellen und Testen einer einfachen Schaltung zur Zustandsänderung per Takt.
- Benötigte Vorkenntnisse: Grundlagen der IEC 61499, Verwendung von Adaptern, logiBUS-Einbindung.

Die Übung eignet sich zum Einstieg in die sequentielle Logik mit Speicherverhalten.

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

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

2.1.2.53 Uebung_004a5_AX: wie Uebung_004a4 aber ohne E_SPLIT



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_004a5_AX`. Ähnlich wie bei `Uebung_004a3_AX` (Impliziter Merge) wird hier gezeigt, dass auch ein Event-Split oft ohne expliziten Baustein möglich ist.

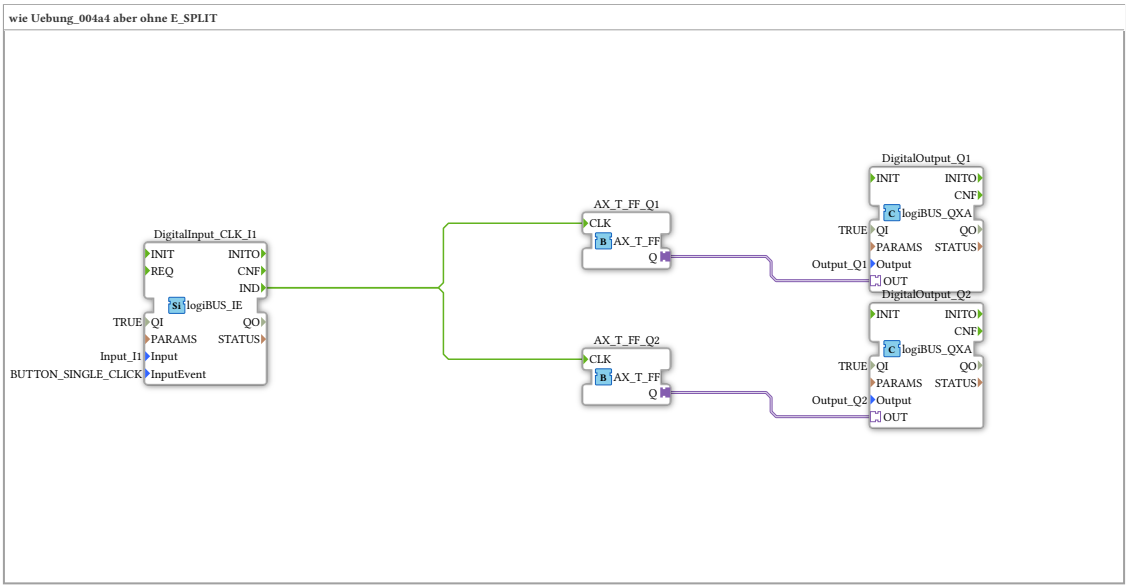
Ziel der Übung

Demonstration der „Fan-Out"-Fähigkeit von Ereignisverbindungen in 4diac. Ein Quell-Event kann mit mehreren Ziel-Events verbunden werden.

Beschreibung und Komponenten

Die Subapplikation `Uebung_004a5_AX.SUB` entfernt den `E_SPLIT` Baustein aus der vorherigen Übung und verbindet den Taster direkt mit beiden Flip-Flops.

Funktionsbausteine (FBs)



- **`DigitalInput_CLK_I1`**: Taster.
- **`E_T_FF_Q1` & `Q2`**: Flip-Flops.

Funktionsweise

```
<EventConnections>
    <Connection Source="DigitalInput_CLK_I1.IND" Destination="E_T_FF_Q1.CLK"/>
    <Connection Source="DigitalInput_CLK_I1.IND" Destination="E_T_FF_Q2.CLK"/>
</EventConnections>
```

Wenn `I1` ein Event feuert, wird dieses an alle verbundenen Ziele verteilt. Die Reihenfolge der Abarbeitung ist in der IEC 61499 Norm nicht strikt für „Fan-Out“ definiert (es ist implementationsabhängig, meistens in der Reihenfolge der Erstellung der Verbindung). Wenn die Reihenfolge kritisch ist, **muss** ein `E_SPLIT` verwendet werden. Wenn es egal ist (wie hier, wo nur zwei Lampen toggeln sollen), reicht die direkte Verbindung.

Anwendungsbeispiel

Gleiches Beispiel wie zuvor (Zentral-Aus), aber platzsparender implementiert.

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

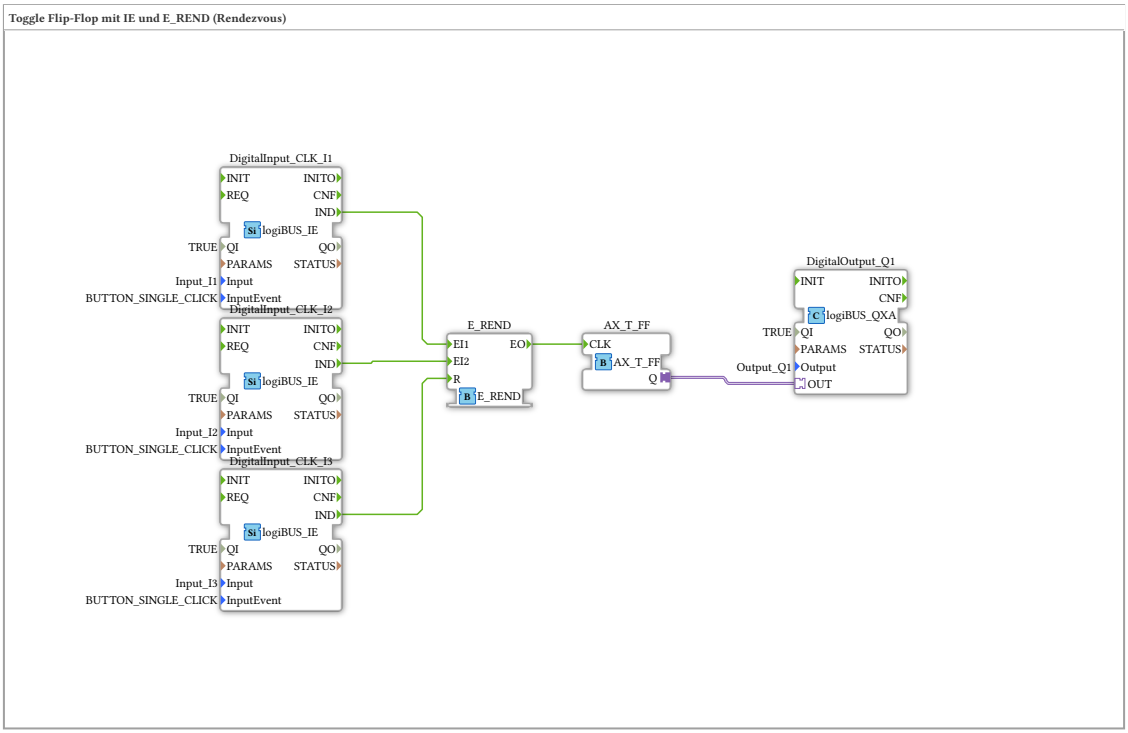
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.1.2.54 Uebung_004a6_AX: Toggle Flip-Flop mit IE und E_REND (Rendezvous)

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_004a6_AX`. Hier wird ein komplexeres Event-Handling-Muster vorgestellt: Das Rendezvous. Ein Ereignis wird erst dann weitergegeben, wenn zwei Bedingungen (Events) eingetreten sind.

Ziel der Übung



Verständnis des `E_REND` Bausteins. Dieser Baustein fungiert wie ein „UND“ für Ereignisse. Er merkt sich, welche Eingänge bereits gefeuert haben, und feuert erst am Ausgang, wenn *alle* erforderlichen Eingänge mindestens einmal aktiv waren. Danach setzt er sich zurück.

Beschreibung und Komponenten

Die Subapplikation `Uebung_004a6_AX.SUB` nutzt `E_REND`, um sicherzustellen, dass zwei Taster gedrückt wurden, bevor das Licht umschaltet.

Funktionsbausteine (FBs)

- **`DigitalInput_CLK_I1` & `I2`**: Die beiden Bestätigungs-Taster.
- **`DigitalInput_CLK_I3`**: Ein Reset-Taster.
- **`E_REND`**: Der Rendezvous-Baustein mit Eingängen `EI1`, `EI2` und einem Reset `R`.
- **`E_T_FF`**: Das Flip-Flop.
- **`DigitalOutput_Q1`**: Die Lampe.

Funktionsweise

```
<EventConnections>
    <Connection Source="DigitalInput_CLK_I1.IND" Destination="E_REND.EI1"/>
    <Connection Source="DigitalInput_CLK_I2.IND" Destination="E_REND.EI2"/>
    <Connection Source="E_REND.EO" Destination="E_T_FF.CLK"/>
    <Connection Source="DigitalInput_CLK_I3.IND" Destination="E_REND.R"/>
</EventConnections>
```

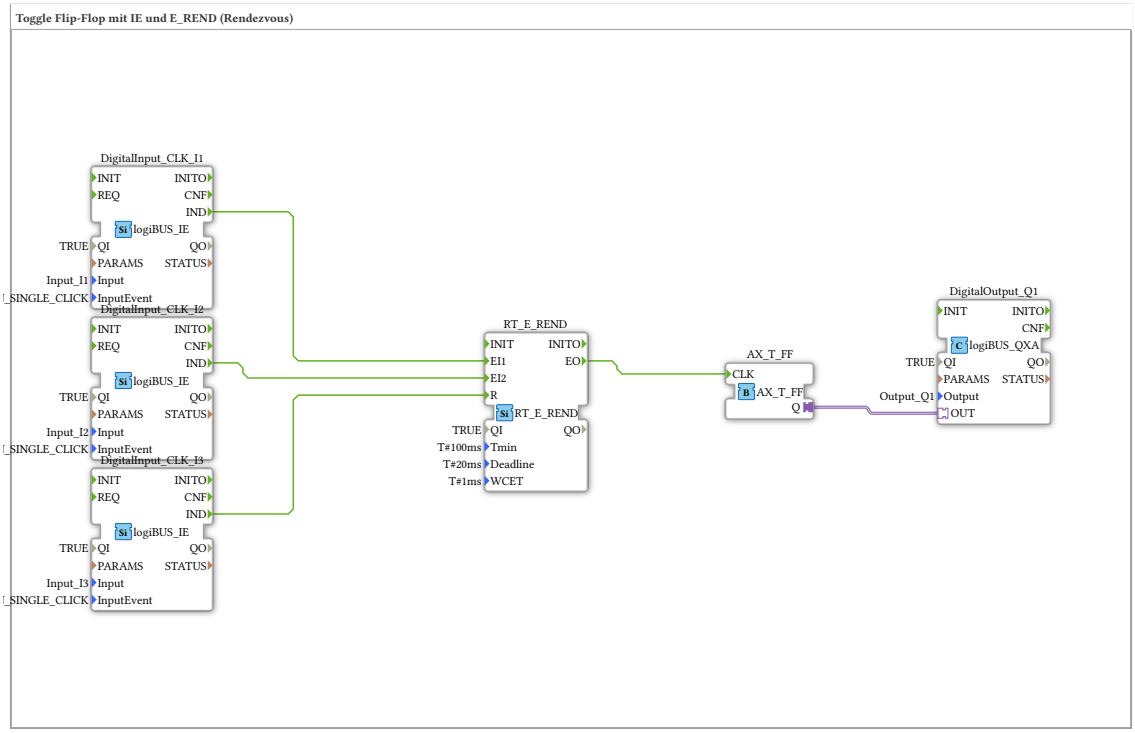
1. Drückt man nur Taster 1 (`I1`), passiert nichts am Ausgang. `E_REND` merkt sich intern „EI1 war da“.
2. Drückt man danach Taster 2 (`I2`), ist die Bedingung komplett (beide waren da). `E_REND` feuert `EO`.
3. Das Flip-Flop schaltet um, die Lampe ändert ihren Zustand.
4. `E_REND` vergisst den Status und wartet erneut auf beide Events.

- Der Reset-Taster (`I3`) kann genutzt werden, um den internen Merker des `E_REND` zu löschen, falls man z.B. nur Taster 1 gedrückt hat und den Vorgang abbrechen will.

Anwendungsbeispiel

Zweihand-Auslösung (Sequenziell): Ein Prozess soll erst starten, wenn Bediener A „Freigabe“ drückt UND Bediener B „Start“ drückt (Reihenfolge egal, aber beide müssen einmal gedrückt haben).

2.1.2.55 Uebung_004a6a_AX: Toggle Flip-Flop mit IE und E_REND (Rendezvous)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung realisiert einen **Toggle-Flip-Flop** (Wechselschalter) unter Verwendung von **Ereignis-Eingängen** (IE) und einem **Rendezvous-Baustein** (`RT_E_REND`). Das System erwartet zwei Tasterereignisse (Eingänge I1 und I2), die innerhalb einer bestimmten Zeitschranke (Deadline) eintreffen müssen. Erst wenn beide Ereignisse synchronisiert wurden, wird der Flip-Flop getaktet und der Digitalausgang Q1 umgeschaltet. Ein dritter Taster (I3) dient als Reset für den Rendezvous-Mechanismus. Die Übung demonstriert den Umgang mit zeitkritischen Ereignisverbindungen, Rendezvous-Synchronisation und einfachen Toggle-Funktionen in der 4diac-IDE.

Verwendete Funktionsbausteine (FBs)

Die SubApp verwendet folgende (Sub-)Bausteine:

- **`DigitalInput_CLK_I1`** – Typ: `logiBUS::io::DI::logiBUS_IE`
  - **Parameter:**
    - `QI` = `TRUE`
    - `Input` = `Input_I1`
    - `InputEvent` = `BUTTON_SINGLE_CLICK`
  - **Ereignisausgang:** `IND` (sendet Ereignis bei Tastendruck)
  - **Funktion:** Erzeugt ein Ereignis, sobald der Taster an Eingang I1 einmal gedrückt wird.
- **`DigitalInput_CLK_I2`** – Typ: `logiBUS::io::DI::logiBUS_IE`
  - **Parameter:**
    - `QI` = `TRUE`
    - `Input` = `Input_I2`
    - `InputEvent` = `BUTTON_SINGLE_CLICK`
  - **Ereignisausgang:** `IND`
  - **Funktion:** Erzeugt ein Ereignis bei Tastendruck an Eingang I2.

- **`DigitalInput_CLK_I3`** – Typ: `logiBUS::io::DI::logiBUS_IE`
  - **Parameter:**
    - `QI` = `TRUE`
    - `Input` = `Input_I3`
    - `InputEvent` = `BUTTON_SINGLE_CLICK`
  - **Ereignisausgang:** `IND`
  - **Funktion:** Erzeugt ein Ereignis bei Tastendruck an Eingang I3 (dient als Reset).
- **`RT_E_REND`** – Typ: `eclipse4diac::rtevents::RT_E_REND`
  - **Parameter:**
    - `QI` = `TRUE`
    - `Tmin` = `T#100ms` (minimale Ereigniszeit, hier nicht ausgenutzt)
    - `Deadline` = `T#20ms` (maximale Zeit zwischen EI1 und EI2)
    - `WCET` = `T#1ms` (worst-case execution time)
  - **Ereigniseingänge:** `EI1`, `EI2`, `R` (Reset)
  - **Ereignisausgang:** `EO`
  - **Datenverbindungen:** keine
  - **Funktion:** Führt ein Rendezvous zwischen den Ereignissen an `EI1` und `EI2` durch. Wenn beide innerhalb der `Deadline` eintreffen, wird `EO` ausgelöst. Der Eingang `R` setzt den internen Zustand zurück.
- **`AX_T_FF`** – Typ: `adapter::events::unidirectional::AX_T_FF`
  - **Parameter:** keine
  - **Ereigniseingang:** `CLK` (Taktsignal)
  - **Adapterausgang:** `Q`
  - **Funktion:** Toggle-Flip-Flop. Jedes Ereignis am `CLK`-Eingang wechselt den Zustand des Ausgangs `Q`.
- **`DigitalOutput_Q1`** – Typ: `logiBUS::io::DQ::logiBUS_QXA`
  - **Parameter:**
    - `QI` = `TRUE`
    - `Output` = `Output_Q1`
  - **Adaptereingang:** `OUT` (Steuersignal)
  - **Funktion:** Gibt den Wert von `OUT` auf dem physischen Ausgang Q1 aus.

Programmablauf und Verbindungen

1. **Ereigniserfassung:**
   - Die drei Tastereingänge (`Input_I1`, `Input_I2`, `Input_I3`) werden durch die `DigitalInput_CLK_IX`-Bausteine überwacht. Bei jedem einfachen Tastendruck (Ereignis `BUTTON_SINGLE_CLICK`) wird der Ereignisausgang `IND` aktiviert.
2. **Rendezvous (Ereignissynchronisation):**
   - Die Ereignisse von `I1` und `I2` werden an `EI1` und `EI2` des `RT_E_REND`-Bausteins weitergeleitet.
   - Solange nicht beide Ereignisse eingetroffen sind, wartet der Baustein. Die maximale Wartezeit zwischen dem ersten und zweiten Ereignis beträgt 20 ms (`Deadline`). Überschreitet die Differenz diesen Wert, wird der Vorgang verworfen und der nächste Versuch wird erwartet.
   - Ein Ereignis von `I3` (Reset-Pin) setzt den Rendezvous-Zustand sofort zurück, ohne `EO` auszulösen.
3. **Toggle-Flip-Flop:**
   - Wenn der Rendezvous erfolgreich ist, sendet `RT_E_REND` ein Ereignis an den `CLK`-Eingang des `AX_T_FF`.
   - Der Flip-Flop wechselt seinen internen Zustand (von `FALSE` auf `TRUE` oder umgekehrt) und gibt diesen über den Adapterausgang `Q` aus.
4. **Ausgabe:**
   - Der Zustand des Flip-Flops (`Q`) wird an den `OUT`-Adaptereingang des `DigitalOutput_Q1`-Bausteins angeschlossen. Dieser steuert den physischen Ausgang `Output_Q1` entsprechend.
   - Bei jedem erfolgreichen Rendezvous schaltet die Ausgabe also um (Toggle-Funktion).

Zusammenfassende Verbindungstabelle:

Quelle	Ziel
DigitalInput_CLK_I1.IND	RT_E_REND.EI1
DigitalInput_CLK_I2.IND	RT_E_REND.EI2
DigitalInput_CLK_I3.IND	RT_E_REND.R
RT_E_REND.EO	AX_T_FF.CLK
AX_T_FF.Q (Adapter)	DigitalOutput_Q1.OUT (Adapter)

Zusammenfassung

Die Übung zeigt:

- Die Verwendung von **Ereignis-Eingängen** (logiBUS_IE) zur Erfassung von Tasterdrücken.
- Die **zeitgesteuerte Rendezvous-Synchronisation** (RT_E_REND) mit konfigurierbarer Deadline.
- Den Betrieb eines **Toggle-Flip-Flops** (AX_T_FF), der durch das Rendezvous-Ereignis getaktet wird.
- Die Anbindung eines **Digitalausgangs** (logiBUS_QXA) zur Ausgabe des Flip-Flop-Zustands.

Damit wird die Grundlage für zeitkritische, ereignisgesteuerte Logik in der Automatisierungstechnik vermittelt.

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

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

2.1.2.56 Uebung_004a7_AX: SR und T-Flip-Flop mit IE und E_REND (Rendezvous)



Dieser Artikel beschreibt die logiBUS®-Übung Uebung_004a7_AX. Sie kombiniert das Rendezvous-Muster mit einem erweiterten Flip-Flop-Typ, der Set- und Reset-Funktionalität bietet.

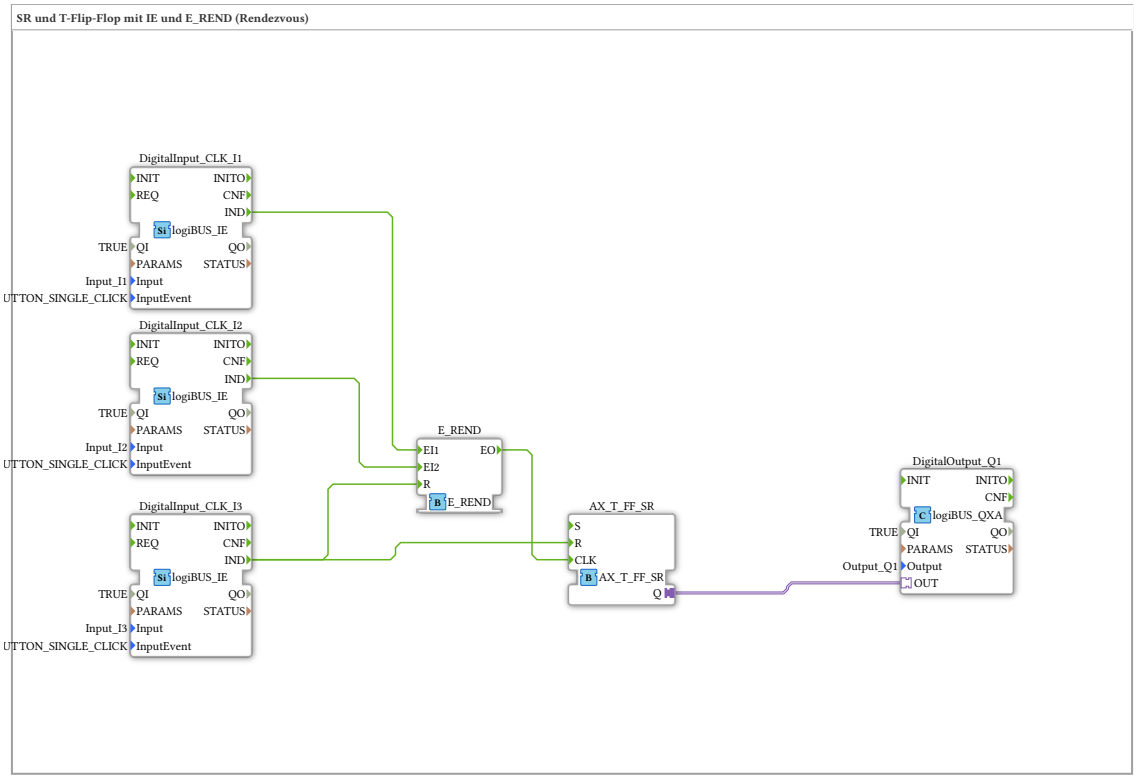
Ziel der Übung

Demonstration der Interaktion zwischen Ereignis-Logik (E_REND) und Zustands-Logik (AX_T_FF_SR - Toggle Flip-Flop mit Set/Reset).

Beschreibung und Komponenten

Die Subapplikation Uebung_004a7_AX.SUB verwendet zwei Taster zum „Scharfschalten“ (Rendezvous) und einen dritten zum expliziten Rücksetzen.

Funktionsbausteine (FBs)



- **`I1` & `I2`**: Eingänge für das Rendezvous.
- **`I3`**: Reset-Eingang.
- **`E_REND`**: Synchronisiert `I1` und `I2`.
- **`AX_T_FF_SR`**: Ein Toggle-Flip-Flop, das zusätzlich einen `R` (Reset) Eingang hat, um den Ausgang definiert auf FALSE zu setzen.

Funktionsweise

```
<EventConnections>
    <Connection Source="DigitalInput_CLK_I1.IND" Destination="E_REND.EI1"/>
    <Connection Source="DigitalInput_CLK_I2.IND" Destination="E_REND.EI2"/>
    <Connection Source="E_REND.EO" Destination="AX_T_FF_SR.CLK"/>
    <Connection Source="DigitalInput_CLK_I3.IND" Destination="AX_T_FF_SR.R"/>
    <Connection Source="DigitalInput_CLK_I3.IND" Destination="E_REND.R"/>
</EventConnections>
```

1. Um die Lampe (`Q1`) einzuschalten (oder umzuschalten), müssen `I1` und `I2` gedrückt werden (Rendezvous -> `CLK`).
2. Der Taster `I3` ist der „Not-Aus“ oder „Alles Löschen“. Er ist verbunden mit:
   - `E_REND.R`: Löscht eventuell halb-fertige Rendezvous-Zustände.
   - `AX_T_FF_SR.R`: Setzt das Flip-Flop hart auf FALSE zurück (Lampe aus).

Anwendungsbeispiel

Maschinenstart mit Reset: Zwei Sicherheitszonen müssen als „frei“ gemeldet werden (`I1`, `I2`), bevor die Maschine starten kann (`CLK`). Ein Not-Halt-Taster (`I3`) stoppt die Maschine sofort und löscht alle Freigaben.

2.1.2.57 Uebung_004a8_AX: mit E_SPLIT_2



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_004a8_AX`. Dies ist eine Variante von `Uebung_004a4_AX`, bei der ein spezifischer `E_SPLIT_2` Baustein verwendet wird, der explizit für 2 Ausgänge gedacht ist.

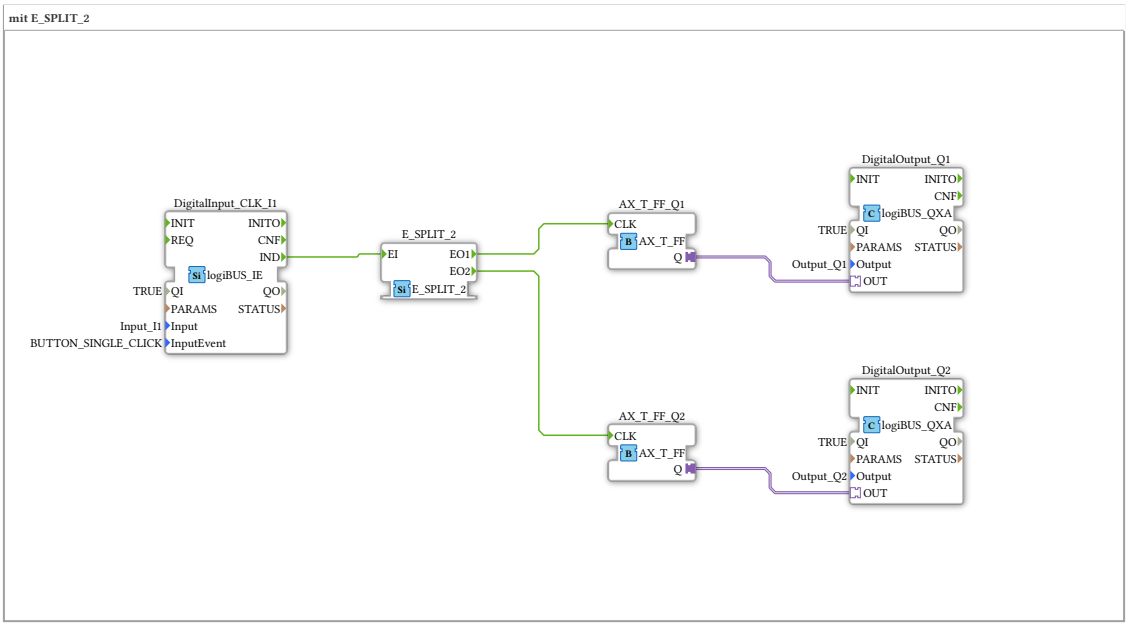
Ziel der Übung

Kennenlernen der spezifischen Splitter-Bausteine. `E_SPLIT` ist oft der generische Name, aber in vielen Bibliotheken gibt es spezifische Versionen wie `E_SPLIT_2`, `E_SPLIT_3` etc., um die Anzahl der Ausgänge festzulegen.

Beschreibung und Komponenten

Die Subapplikation `Uebung_004a8_AX.SUB` nutzt `E_SPLIT_2`, um einen Tasterklick auf zwei unabhängige Flip-Flops zu verteilen.

Funktionsbausteine (FBs)



- **`DigitalInput_CLK_I1`**: Taster.
- **`E_SPLIT_2`**: Verteilt Eingang `EI` sequenziell auf `EO1` und `EO2`.
- **`AX_T_FF_Q1` & `Q2`**: Zwei Flip-Flops für die Ausgänge `Q1` und `Q2`.

Funktionsweise

```
<EventConnections>
    <Connection Source="DigitalInput_CLK_I1.IND" Destination="E_SPLIT_2.EI"/>
    <Connection Source="E_SPLIT_2.EO1" Destination="AX_T_FF_Q1.CLK"/>
    <Connection Source="E_SPLIT_2.EO2" Destination="AX_T_FF_Q2.CLK"/>
</EventConnections>
```

Funktional identisch zu `Uebung_004a4_AX`: Ein Eingangsevent löst nacheinander zwei Ausgangsevents aus, wodurch beide Flip-Flops sicher und definiert angesteuert werden.

Anwendungsbeispiel

Synchrones Schalten von redundanten Systemen, bei denen sichergestellt sein muss, dass beide Systeme den Schaltbefehl erhalten.

2.1.2.58 Uebung_004a9_AX: mit E_SPLIT_3

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_004a9_AX`. Hier wird das Konzept des Event-Splittings auf drei Ziele erweitert.

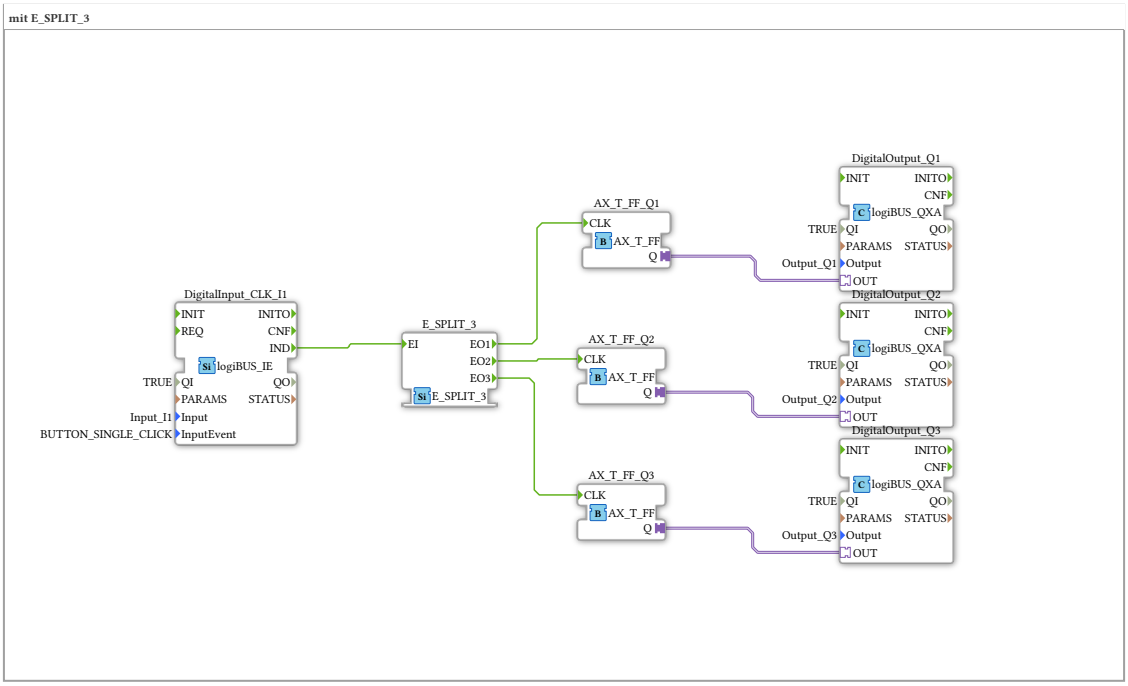
Ziel der Übung

Demonstration der Skalierbarkeit von Event-Verteilern. Mit `E_SPLIT_3` können drei Prozesse sequenziell angestoßen werden.

Beschreibung und Komponenten

Die Subapplikation `Uebung_004a9_AX.SUB` verteilt das Signal eines Tasters auf drei separate Toggle-Flip-Flops und somit auf drei Ausgänge.

Funktionsbausteine (FBs)



- **`DigitalInput_CLK_I1`**: Taster.
- **`E_SPLIT_3`**: Verteilt Eingang `EI` sequenziell auf `EO1`, `EO2` und `EO3`.
- **`AX_T_FF_Q1, Q2, Q3`**: Drei Flip-Flops.
- **`DigitalOutput_Q1, Q2, Q3`**: Drei Lampen.

Funktionsweise

```
<EventConnections>
    <Connection Source="DigitalInput_CLK_I1.IND" Destination="E_SPLIT_3.EI"/>
    <Connection Source="E_SPLIT_3.EO1" Destination="AX_T_FF_Q1.CLK"/>
    <Connection Source="E_SPLIT_3.EO2" Destination="AX_T_FF_Q2.CLK"/>
    <Connection Source="E_SPLIT_3.EO3" Destination="AX_T_FF_Q3.CLK"/>
</EventConnections>
```

Ein einzelner Klick auf den Taster löst eine Kaskade aus:

1. `EO1` feuert -> `Q1` toggelt.
2. `EO2` feuert -> `Q2` toggelt.
3. `EO3` feuert -> `Q3` toggelt.

Dies geschieht in der SPS-Zykluszeit so schnell, dass es für das menschliche Auge simultan wirkt, aber steuerungstechnisch ist es eine definierte Sequenz.

Anwendungsbeispiel

Zentral-Schalter für eine Etage: Ein Taster an der Wohnungstür schaltet Licht im Flur (`Q1`), Küche (`Q2`) und Wohnzimmer (`Q3`) gleichzeitig aus (oder um).

2.1.2.59 Uebung_004a_AX: Toggle Flip-Flop mit IE mit BUTTON_SINGLE_CLICK



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_004a_AX`. In dieser Übung verlassen wir die reine Datenweiterleitung und nutzen Ereignisse (Events), um eine Speicherfunktion zu realisieren: einen klassischen Stromstoßschalter.

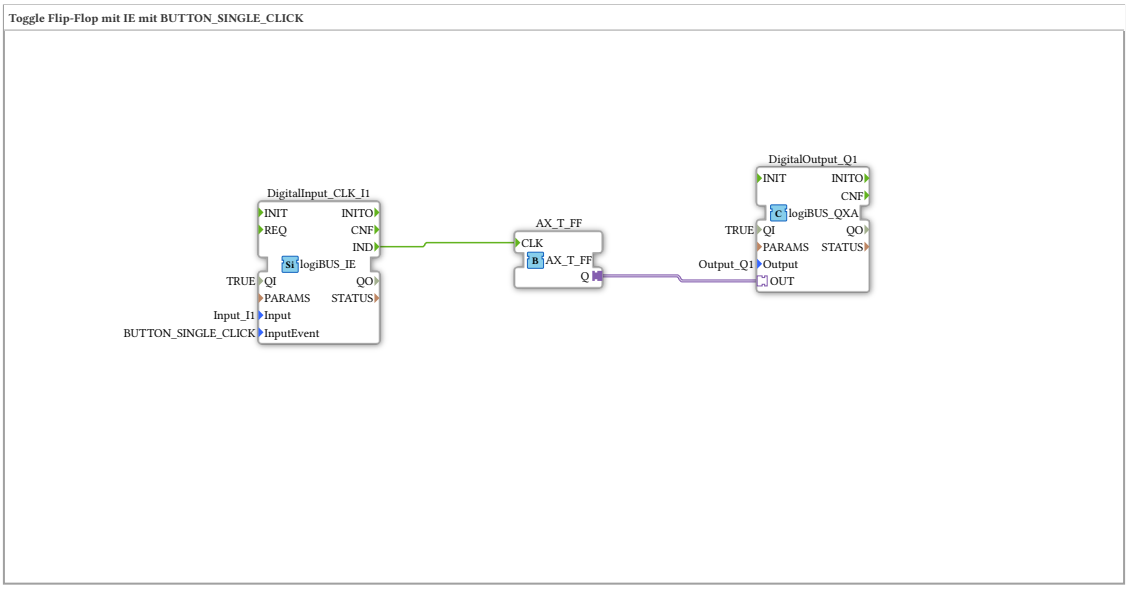
Ziel der Übung

Das Ziel ist es, den Unterschied zwischen zustandsorientierter (Pegel) und ereignisorientierter (Flanke) Programmierung zu verstehen. Während ein einfacher Taster nur solange „Ein“ ist, wie er gedrückt wird, soll hier jeder Tastendruck den Zustand des Ausgangs wechseln (Umschalten: Aus -> Ein -> Aus -> ...).

Beschreibung und Komponenten

Die Subapplikation `Uebung_004a_AX.SUB` verwendet einen speziellen Eingangsbaustein, der Klick-Ereignisse generiert, und ein Toggle-Flip-Flop.

Funktionsbausteine (FBs)



- **`DigitalInput_CLK_I1`**: Typ `logiBUS_IE` (Input Event). Im Gegensatz zum `IXA` (Input Extended Adapter) liefert dieser Baustein kein kontinuierliches `BOOL`-Signal, sondern feuert ein einzelnes Ereignis (`IND`), wenn eine bestimmte Bedingung erfüllt ist. Hier ist er auf `BUTTON_SINGLE_CLICK` konfiguriert.
- **`E_T_FF`**: Typ `AX_T_FF` (Adapter Toggle Flip-Flop). Dieser Baustein hat einen Takteingang (`CLK`). Bei jedem empfangenen Ereignis wechselt er seinen internen Zustand und gibt diesen über den Adapter-Ausgang `Q` aus.
- **`DigitalOutput_Q1`**: Typ `logiBUS_QXA`. Schaltet den physischen Ausgang `Q1` basierend auf dem Zustand des Flip-Flops.

Funktionsweise

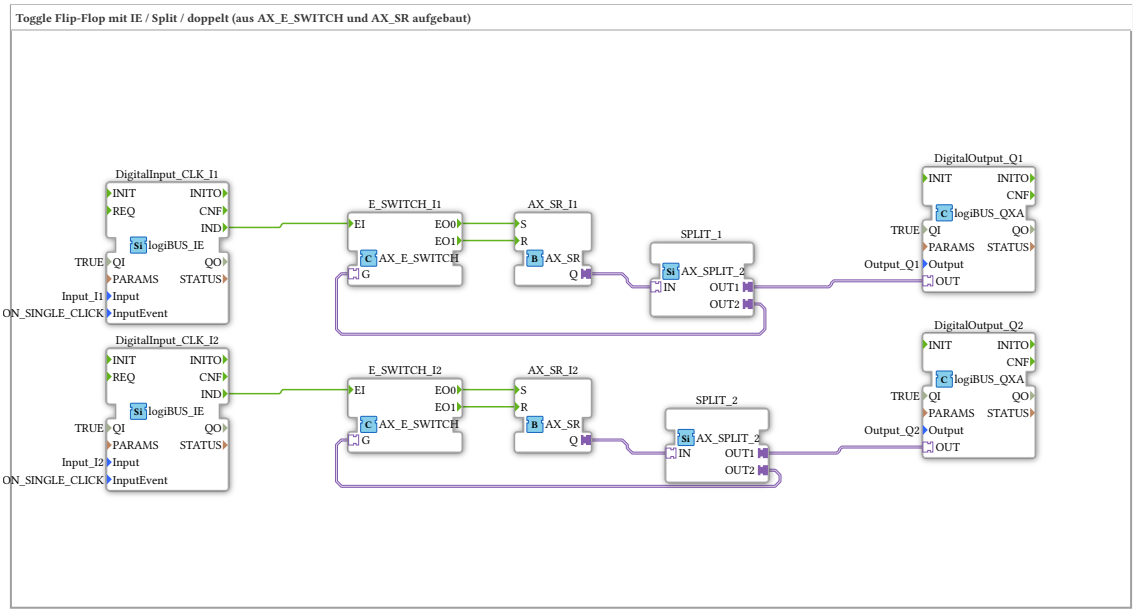
1. Der Benutzer drückt den Taster an `I1` kurz („Klick“).
2. Der `DigitalInput_CLK_I1` erkennt das Muster „Einzelklick“ und sendet ein `IND`-Ereignis.
3. Das Ereignis erreicht den `CLK`-Eingang des `E_T_FF`.
4. Das Flip-Flop kippt seinen Zustand (z.B. von FALSE auf TRUE).
5. Der neue Zustand wird über den Adapter-Ausgang `Q` an `DigitalOutput_Q1` gesendet.
6. Die Lampe an `Q1` geht an und bleibt an, auch wenn der Taster losgelassen wird.
7. Beim nächsten Klick wiederholt sich der Vorgang, das Flip-Flop kippt zurück auf FALSE, die Lampe geht aus.

Anwendungsbeispiel

Die klassische **Flurbeleuchtung** oder **Treppenhauslicht** (ohne Zeitglied): Ein Tasterdruck schaltet das Licht ein, der nächste schaltet es wieder aus. Dies ist mit einem rein elektrischen Schalter (der zurückfedert) nicht möglich, man benötigt ein Speicherelement (Stromstoßrelais in der Elektrotechnik, Flip-Flop in der Software).

2.1.2.60 Uebung_004b2_AX: Toggle Flip-Flop mit IE / Split / doppelt (aus AX_E_SWITCH und AX_SR aufgebaut)

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_004b2_AX (Toggle Flip-Flop mit IE / Split / doppelt (aus AX_E_SWITCH und AX_SR aufgebaut)).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Toggle Flip-Flop mit IE / Split / doppelt (aus AX_E_SWITCH und AX_SR aufgebaut)**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_004b2_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
- **DigitalInput_CLK_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **E_SWITCH_I1**: Instanz des Typs adapter::events::unidirectional::AX_E_SWITCH.
- **AX_SR_I1**: Instanz des Typs adapter::events::unidirectional::AX_SR.
- **DigitalOutput_Q2**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q2
- **DigitalInput_CLK_I2**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I2
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **E_SWITCH_I2**: Instanz des Typs adapter::events::unidirectional::AX_E_SWITCH.

- **AX_SR_I2**: Instanz des Typs adapter::events::unidirectional::AX_SR.
- **SPLIT_1**: Instanz des Typs adapter::events::unidirectional::AX_SPLIT_2.
- **SPLIT_2**: Instanz des Typs adapter::events::unidirectional::AX_SPLIT_2.

Verbindungen und Schnittstellen

Adapterverbindungen:

- AX_SR_I1.Q -> SPLIT_1.IN
- SPLIT_1.OUT2 -> E_SWITCH_I1.G
- SPLIT_2.OUT2 -> E_SWITCH_I2.G
- AX_SR_I2.Q -> SPLIT_2.IN
- SPLIT_2.OUT1 -> DigitalOutput_Q2.OUT
- SPLIT_1.OUT1 -> DigitalOutput_Q1.OUT

Ereignisverbindungen:

- DigitalInput_CLK_I1.IND -> E_SWITCH_I1.EI
- E_SWITCH_I1.EO0 -> AX_SR_I1.S
- E_SWITCH_I1.EO1 -> AX_SR_I1.R
- DigitalInput_CLK_I2.IND -> E_SWITCH_I2.EI
- E_SWITCH_I2.EO0 -> AX_SR_I2.S
- E_SWITCH_I2.EO1 -> AX_SR_I2.R

Programmablauf und Funktionsweise

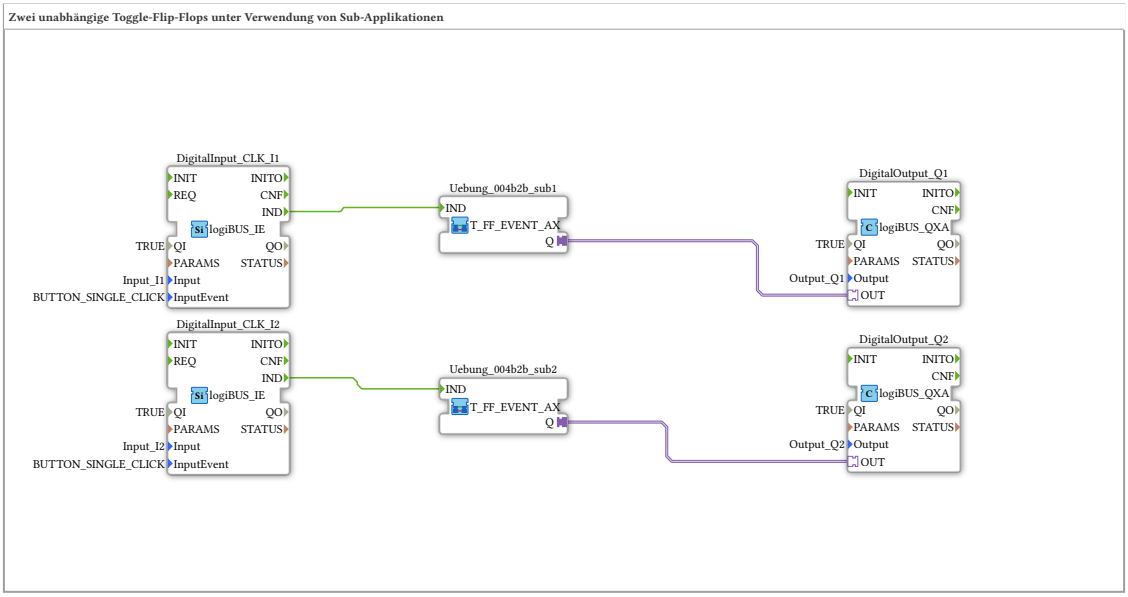
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_004b2_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.61 Uebung_004b2b_AX: Zwei unabhängige Toggle-Flip-Flops unter Verwendung von Sub-Applikationen

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_004b2b_AX (Zwei unabhängige Toggle-Flip-Flops unter Verwendung von Sub-Applikationen).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Zwei unabhängige Toggle-Flip-Flops unter Verwendung von Sub-Applikationen**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_004b2b_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
- **DigitalInput_CLK_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **DigitalOutput_Q2**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q2
- **DigitalInput_CLK_I2**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I2
  - Parameter InputEvent = BUTTON_SINGLE_CLICK

Verbindungen und Schnittstellen

Adapterverbindungen:

- Uebung_004b2b_sub1.Q -> DigitalOutput_Q1.OUT
- Uebung_004b2b_sub2.Q -> DigitalOutput_Q2.OUT

Ereignisverbindungen:

- DigitalInput_CLK_I1.IND -> Uebung_004b2b_sub1.IND
- DigitalInput_CLK_I2.IND -> Uebung_004b2b_sub2.IND

Programmablauf und Funktionsweise

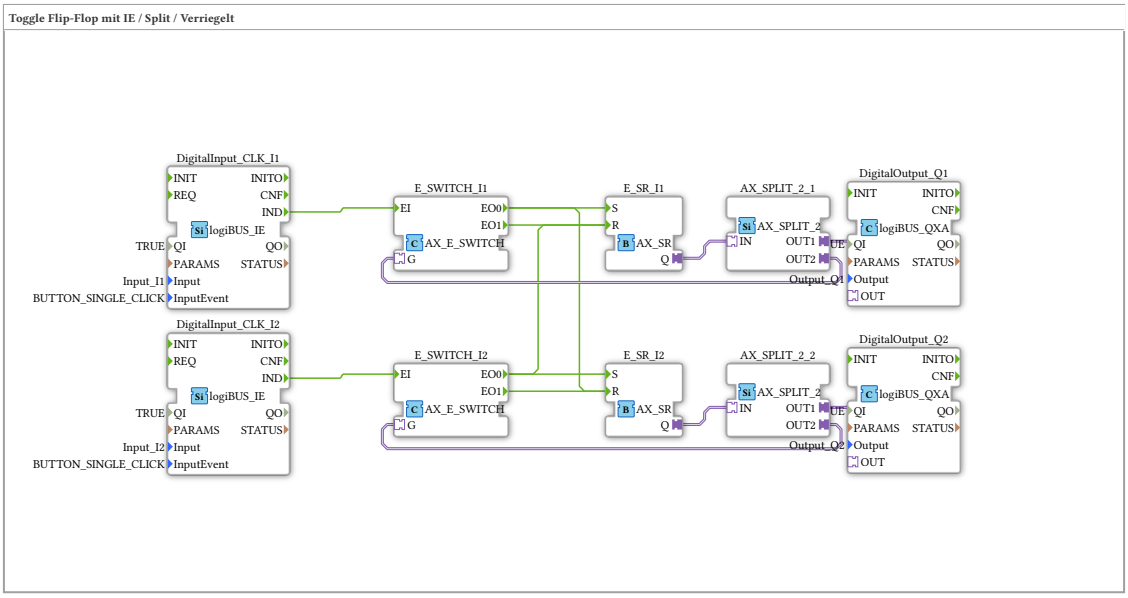
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_004b2b_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.62 Uebung_004b3_AX: Toggle Flip-Flop mit IE / Split / Verriegelt

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_004b3_AX (Toggle Flip-Flop mit IE / Split / Verriegelt).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Toggle Flip-Flop mit IE / Split / Verriegelt**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_004b3_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
- **DigitalInput_CLK_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **E_SR_I1**: Instanz des Typs adapter::events::unidirectional::AX_SR.
- **E_SWITCH_I1**: Instanz des Typs adapter::events::unidirectional::AX_E_SWITCH.
- **E_SWITCH_I2**: Instanz des Typs adapter::events::unidirectional::AX_E_SWITCH.
- **DigitalOutput_Q2**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q2
- **E_SR_I2**: Instanz des Typs adapter::events::unidirectional::AX_SR.
- **AX_SPLIT_2_1**: Instanz des Typs adapter::events::unidirectional::AX_SPLIT_2.
- **AX_SPLIT_2_2**: Instanz des Typs adapter::events::unidirectional::AX_SPLIT_2.
- **DigitalInput_CLK_I2**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I2
  - Parameter InputEvent = BUTTON_SINGLE_CLICK

Verbindungen und Schnittstellen

Adapterverbindungen:

- E_SR_I1.Q -> AX_SPLIT_2_1.IN
- AX_SPLIT_2_1.OUT1 -> DigitalOutput_Q1.OUT
- AX_SPLIT_2_1.OUT2 -> E_SWITCH_I1.G
- E_SR_I2.Q -> AX_SPLIT_2_2.IN
- AX_SPLIT_2_2.OUT1 -> DigitalOutput_Q2.OUT

- AX_SPLIT_2_2.OUT2 -> E_SWITCH_I2.G

Ereignisverbindungen:

- DigitalInput_CLK_I1.IND -> E_SWITCH_I1.EI
- E_SWITCH_I1.EO0 -> E_SR_I1.S
- E_SWITCH_I1.EO1 -> E_SR_I1.R
- E_SWITCH_I2.EO1 -> E_SR_I2.R
- E_SWITCH_I2.EO0 -> E_SR_I2.S
- DigitalInput_CLK_I2.IND -> E_SWITCH_I2.EI
- E_SWITCH_I2.EO0 -> E_SR_I1.R
- E_SWITCH_I1.EO0 -> E_SR_I2.R

Programmablauf und Funktionsweise

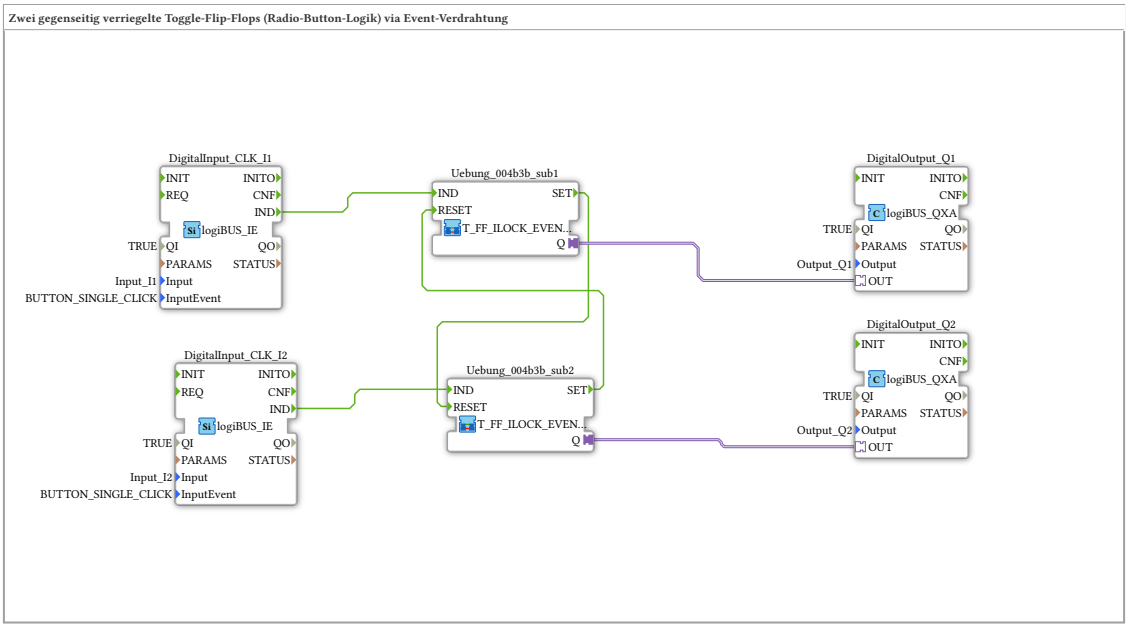
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_004b3_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.63 Uebung_004b3b_AX: Zwei gegenseitig verriegelte Toggle-Flip-Flops (Radio-Button-Logik) via Event-Verdrahtung

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_004b3b_AX (Zwei gegenseitig verriegelte Toggle-Flip-Flops (Radio-Button-Logik) via Event-Verdrahtung).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Zwei gegenseitig verriegelte Toggle-Flip-Flops (Radio-Button-Logik) via Event-Verdrahtung**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_004b3b_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
- **DigitalInput_CLK_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **DigitalOutput_Q2**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q2
- **DigitalInput_CLK_I2**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I2
  - Parameter InputEvent = BUTTON_SINGLE_CLICK

Verbindungen und Schnittstellen

Adapterverbindungen:

- Uebung_004b3b_sub1.Q -> DigitalOutput_Q1.OUT
- Uebung_004b3b_sub2.Q -> DigitalOutput_Q2.OUT

Ereignisverbindungen:

- DigitalInput_CLK_I2.IND -> Uebung_004b3b_sub2.IND
- DigitalInput_CLK_I1.IND -> Uebung_004b3b_sub1.IND
- Uebung_004b3b_sub2.SET -> Uebung_004b3b_sub1.RESET
- Uebung_004b3b_sub1.SET -> Uebung_004b3b_sub2.RESET

Programmablauf und Funktionsweise

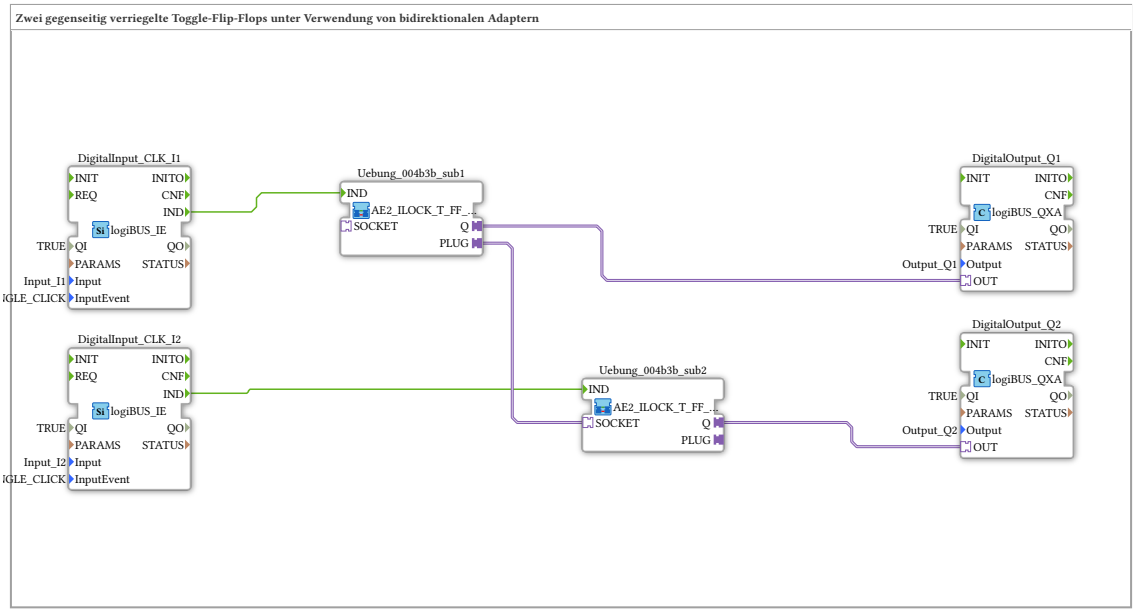
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_004b3b_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.64 Uebung_004b3c_AX: Zwei gegenseitig verriegelte Toggle-Flip-Flops unter Verwendung von bidirektionalen Adaptern

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_004b3c_AX (Zwei gegenseitig verriegelte Toggle-Flip-Flops unter Verwendung von bidirektionalen Adaptern).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Zwei gegenseitig verriegelte Toggle-Flip-Flops unter Verwendung von bidirektionalen Adaptern**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_004b3c_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
- **DigitalInput_CLK_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **DigitalOutput_Q2**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q2
- **DigitalInput_CLK_I2**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I2
  - Parameter InputEvent = BUTTON_SINGLE_CLICK

Verbindungen und Schnittstellen

Adapterverbindungen:

- Uebung_004b3b_sub1.PLUG -> Uebung_004b3b_sub2.SOCKET
- Uebung_004b3b_sub1.Q -> DigitalOutput_Q1.OUT
- Uebung_004b3b_sub2.Q -> DigitalOutput_Q2.OUT

Ereignisverbindungen:

- DigitalInput_CLK_I2.IND -> Uebung_004b3b_sub2.IND
- DigitalInput_CLK_I1.IND -> Uebung_004b3b_sub1.IND

Hinweise aus dem Modell

durch den Einsatz eines Bidirektionalen Adapters: 1 Verbindung REICHT !

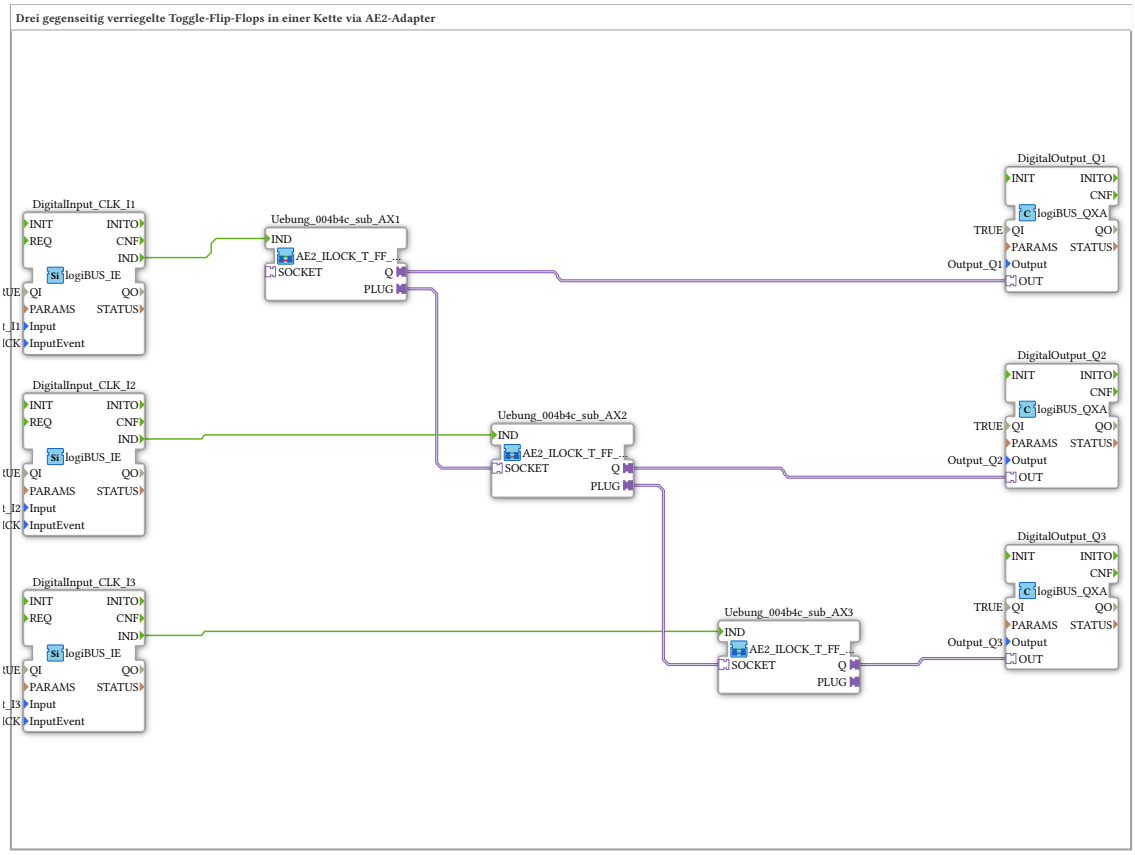
Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_004b3c_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.65 Uebung_004b4c_AX: Drei gegenseitig verriegelte Toggle-Flip-Flops in einer Kette via AE2-Adapter



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung befasst sich mit der Realisierung einer Kette aus drei gegenseitig verriegelten Toggle-Flip-Flops. Die Verriegelung erfolgt über AE2-Adapter (bidirektionale Schnittstellen), sodass jeder Sub-Baustein seinen Zustand nur dann wechseln kann, wenn die vorgängigen Flip-Flops inaktiv sind. Dadurch wird sichergestellt, dass zu jedem Zeitpunkt nur ein Ausgang aktiv sein kann. Die Eingänge werden über logiBUS-Taster (Single-Click-Ereignis) angesteuert, die Ausgänge über logiBUS-LEDs ausgegeben.

Verwendete Funktionsbausteine (FBs)

Die folgende Tabelle listet die im Netzwerk verwendeten Bausteine auf:

Bausteinname	Typ	Beschreibung

`DigitalInput_CLK_I1, I2, I3`	`logiBUS_IE`	Digitaleingang für Taster (Single-Click) an den physikalischen Eingängen `Input_I1`, `Input_I2`, `Input_I3`.
`DigitalOutput_Q1, Q2, Q3`	`logiBUS_QXA`	Digitalausgang zur Ansteuerung der physikalischen Ausgänge `Output_Q1`, `Output_Q2`, `Output_Q3`.
`Uebung_004b4c_sub_AX1 ... AX3`	`Uebung_004b4c_sub_AE` (SubApp)	Sub-Baustein, der jeweils ein Toggle-Flip-Flop mit Verriegelungslogik enthält.

Sub-Bausteine: `Uebung_004b4c_sub_AE`

- **Typ**: SubApp (wiederverwendbare Komponente)
- **Verwendete interne FBs**: Die SubApp implementiert ein Toggle-Flip-Flop (z. B. mit einem SR-Flip-Flop oder einem speichernden Element) sowie eine Verriegelungsschaltung, die den Zustand der benachbarten SubApps über die AE2-Adapter auswertet.
- **Schnittstellen**:
  - Ereigniseingang `IND` (vom Taster)
  - Adapter-Socket und Adapter-Plug (vom Typ AE2) für die bidirektionale Kommunikation mit den Nachbarn
  - Datenausgang `Q` (Bool) für den aktuellen Flip-Flop-Zustand
- **Funktionsweise**:
  Jeder Sub-Baustein arbeitet als Toggle-Flip-Flop: Bei jedem positiven Ereignis an `IND` wechselt der interne Zustand (und damit `Q`), sofern die Verriegelungsbedingung erfüllt ist. Die Verriegelung stellt sicher, dass ein Flip-Flop nur dann umschalten kann, wenn alle vorherigen Flip-Flops in der Kette einen inaktiven Zustand (`Q=0`) aufweisen. Die AE2-Adapter ermöglichen dabei eine unidirektionale Weitergabe des Zustands nach vorne in der Kette. Der Kommentar **„durch den Einsatz eines Bidirektionalen Adapters: 1 Verbindung REICHT!“** weist darauf hin, dass für die gesamte Verriegelungslogik pro SubApp nur eine einzige Adapterverbindung (zum nächsten Glied) notwendig ist.

Programmablauf und Verbindungen

Die drei Sub-Bausteine sind in einer Kette angeordnet:
`Uebung_004b4c_sub_AX1` (erstes Glied) → `Uebung_004b4c_sub_AX2` (zweites Glied) → `Uebung_004b4c_sub_AX3` (drittes Glied)

- **Ereignisverbindungen**:
  Jeder Taster (`DigitalInput_CLK_I1 ... I3`) erzeugt bei Betätigung ein Ereignis am Ausgang `IND`, das direkt an den Ereigniseingang `IND` des zugehörigen Sub-Bausteins weitergeleitet wird.
- **Adapterverbindungen**:
  - Der Plug von `Uebung_004b4c_sub_AX1` ist mit dem Socket von `Uebung_004b4c_sub_AX2` verbunden.
  - Der Plug von `Uebung_004b4c_sub_AX2` ist mit dem Socket von `Uebung_004b4c_sub_AX3` verbunden.
    Über diese Verbindungen wird der aktuelle Zustand des jeweiligen Flip-Flops an den nächsten weitergegeben (z. B. als Sperrsignal). Dadurch kann ein Flip-Flop nur dann toggeln, wenn alle vorherigen Flip-Flops den Zustand `false` liefern.
- **Ausgangsverbindungen**:
  Der Ausgang `Q` jedes Sub-Bausteins ist über eine Adapterverbindung (z. B. `Uebung_004b4c_sub_AX1.Q → DigitalOutput_Q1.OUT`) mit dem entsprechenden Digitalausgang `OUT` des `logiBUS_QXA`-Bausteins verbunden. Die Ausgänge `Q1`, `Q2`, `Q3` werden an die physikalischen LEDs ausgegeben.

Ablauf:

1. Kein Taster betätigt: Alle Ausgänge sind aus (inaktiv).
2. Wird Taster I1 betätigt, toggelt SubApp AX1 auf aktiv (Q1 an). Die Verriegelung erlaubt dies, da alle vorherigen Glieder inaktiv sind.
3. Wird nun Taster I2 betätigt, toggelt SubApp AX2 nur dann, wenn AX1 inaktiv ist. Da AX1 aktuell aktiv ist, erfolgt kein Umschalten (Verriegelung).

4. Erst wenn Taster I1 erneut betätigt wird (AX1 wird inaktiv), kann Taster I2 das zweite Flip-Flop aktivieren.
5. Analog gilt dies für das dritte Glied.

Die Übung veranschaulicht die Realisierung einer sequentiellen Verriegelungskette mit minimalen Verbindungen durch den Einsatz bidirektionaler AE2-Adapter.

Zusammenfassung

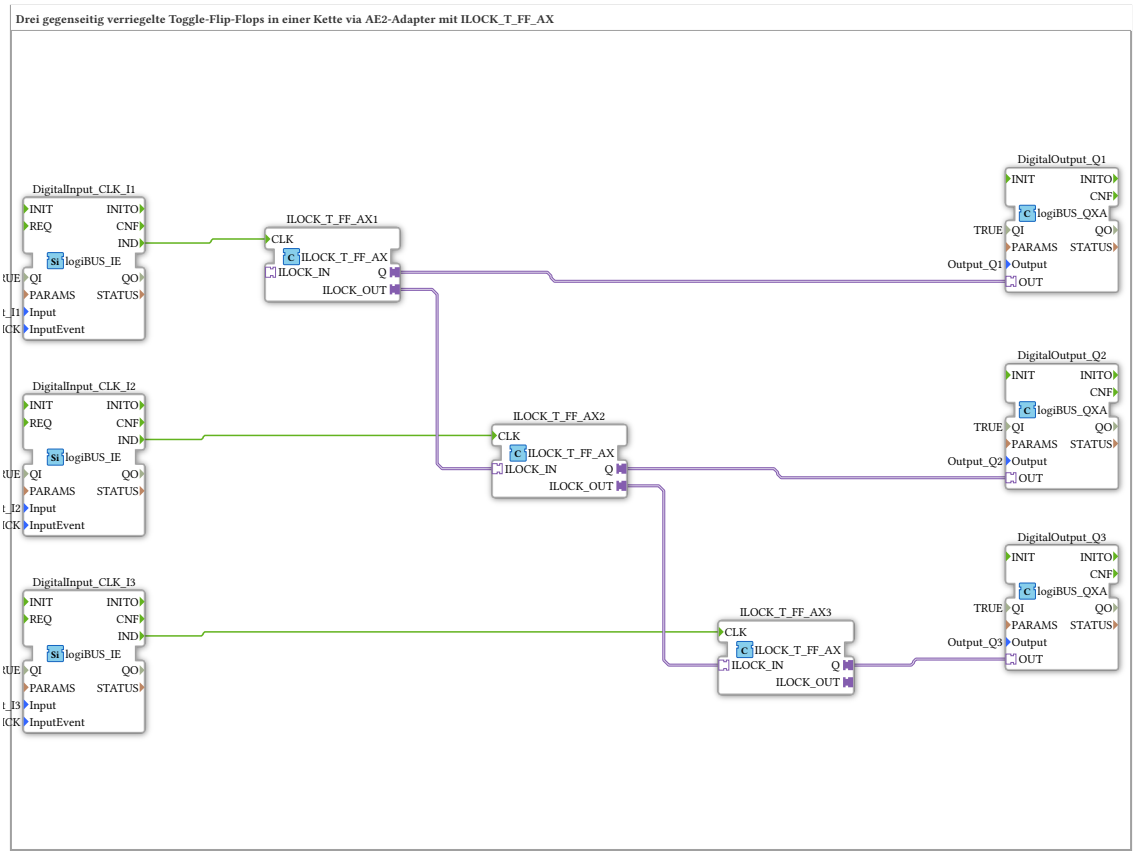
In dieser Übung wurde mithilfe von drei identischen Sub-Bausteinen (`Uebung_004b4c_sub_AE`) eine Kette gegenseitig verriegelter Toggle-Flip-Flops aufgebaut. Die Verriegelung stellt sicher, dass immer nur ein Ausgang aktiv sein kann. Die Umsetzung nutzt die logiBUS-Eingangs- und Ausgangsbausteine sowie AE2-Adapter, um die Kommunikation zwischen den Sub-Bausteinen mit nur einer Kabelverbindung pro Stufe zu realisieren. Auf diese Weise wird ein einfaches und effizientes Sperrlogik-Konzept demonstriert.

2.1.2.66 Uebung_004b4c_sub_AE

Hinweis

Diese Übung existiert nicht mehr als eigene Übung. Der Baustein wurde in die Bibliothek verschoben und heißt jetzt AE2_ILOCK_T_FF_TO_AX. Beschreibung, Schnittstelle und Netzwerk stehen dort.

2.1.2.67 Uebung_004b4d_AX: Drei gegenseitig verriegelte Toggle-Flip-Flops in einer Kette via AE2-Adapter mit ILOCK_T_FF_AX



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung implementiert drei gegenseitig verriegelte Toggle-Flip-Flops, die über einen AE2-Adapter in einer Kette miteinander verbunden sind. Der Funktionsbaustein `ILOCK_T_FF_AX` ermöglicht es, durch Tastendruck (Single-Click) einen Ausgang zu togglen. Dabei werden automatisch alle anderen Ausgänge zurückgesetzt (Verriegelung). Die Kette sorgt dafür, dass immer nur ein Ausgang aktiv sein kann.

Verwendete Funktionsbausteine (FBs)

- **Digital Inputs (logiBUS_IE)** – drei Eingänge für Taster (Input_I1, Input_I2, Input_I3) mit Ereignis BUTTON_SINGLE_CLICK.
- **Digital Outputs (logiBUS_QXA)** – drei Ausgänge (Output_Q1, Output_Q2, Output_Q3) für die Anzeige der aktiven Flip-Flop-Stufe.
- **ILOCK_T_FF_AX** (logiBUS::signalprocessing::interlock::ILOCK_T_FF_AX) – dreimal verwendet:
  - **Funktionsweise**: Ein Bistabiles Element (Toggle-Flip-Flop) mit einem Takt-Eingang `CLK` und einem Ausgang `Q`. Der interne Zustand wird bei jedem CLK-Ereignis umgeschaltet. Über die Adapter `ILOCK_IN` und `ILOCK_OUT` wird eine gegenseitige Verriegelung realisiert: Sobald dieser FB aktiv wird, setzt er über `ILOCK_OUT` alle nachfolgenden FBs zurück, gleichzeitig sperrt `ILOCK_IN` den FB, falls ein vorgelagerter FB aktiv ist.

Sub-Bausteine

Es sind keine Sub-Bausteine innerhalb dieser SubApp definiert. Alle verwendeten FB-Typen stammen aus der Bibliothek `logiBUS`.

Programmablauf und Verbindungen

1. **Ereignisverkettung**
   - Die drei Taster (`DigitalInput_CLK_I1`, `I2`, `I3`) erzeugen bei einem Single-Click ein Ereignis `IND`.
   - Dieses Ereignis wird direkt an den CLK-Eingang des zugehörigen `ILOCK_T_FF_AX` weitergeleitet.
2. **Adapter-Verbindungen (Verriegelungskette)**
   - `ILOCK_T_FF_AX1.ILOCK_OUT` ⟶ `ILOCK_T_FF_AX2.ILOCK_IN`
   - `ILOCK_T_FF_AX2.ILOCK_OUT` ⟶ `ILOCK_T_FF_AX3.ILOCK_IN`
   - Dadurch entsteht eine Kaskade: Wird FB1 aktiv, sperrt es FB2; wird FB2 aktiv, sperrt es FB3. Ein nachfolgender FB kann nur aktiv werden, wenn der vorherige inaktiv ist.
   - Ein Kommentar im Netzwerk weist darauf hin, dass durch den bidirektionalen Adapter nur eine Verbindung pro Stufe ausreicht.
3. **Ausgangsverkettung**
   - Jeder `ILOCK_T_FF_AX` hat einen Ausgangs-Adapter `Q`, der an den entsprechenden Digital-Ausgang (`DigitalOutput_Q1`…`Q3`) angeschlossen ist.
   - Die Ausgänge zeigen den Zustand des aktiven Flip-Flops an.
4. **Ablauf**
   - Durch Drücken von Taster I1, I2 oder I3 wird das jeweilige Flip-Flop umgeschaltet.
   - Wenn ein Flip-Flop aktiv wird, werden alle nachfolgenden Flip-Flops in der Kette zurückgesetzt.
   - Ein vorher aktives Flip-Flop bleibt nur aktiv, solange kein vorgelagertes Flip-Flop getoggelt wird.

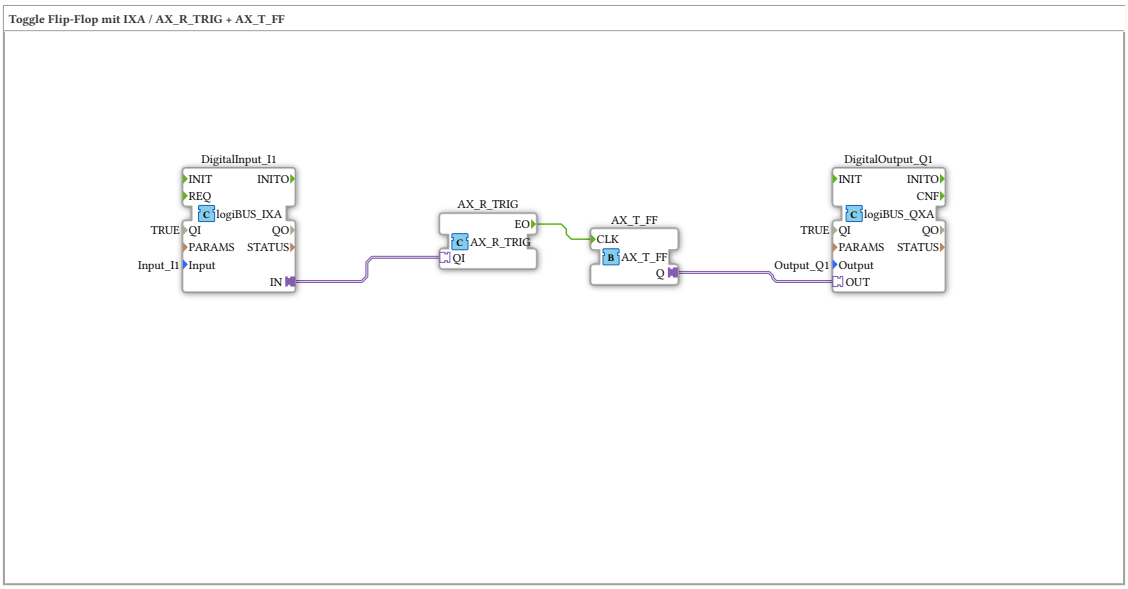
Zusammenfassung

Die Übung demonstriert den Aufbau einer verriegelten Toggle-Kette mit Hilfe des Funktionsbausteins `ILOCK_T_FF_AX`. Lernziele sind das Verständnis von:

- Toggle-Flip-Flops und deren Zustandswechsel,
- gegenseitiger Verriegelung (Interlocking) über Adapter-Schnittstellen,
- kaskadierter Sperrlogik, bei der immer nur genau ein Ausgang aktiv sein kann.

Der Aufbau eignet sich für Anwendungen wie Umschalt- oder Prioritätssteuerungen, bei denen mehrere Bedienelemente exklusiv wirken sollen.

2.1.2.68 Uebung_004b5_AX: Toggle Flip-Flop mit IXA / AX_R_TRIG + AX_T_FF



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung realisiert einen Toggle-Flipflop unter Verwendung eines logiBUS-Eingangs- und Ausgangs-Adapters (IXA/QXA) sowie der Adapter-Funktionsbausteine AX_R_TRIG (Flankendetektion) und AX_T_FF (Toggle-Flipflop). Das Verhalten: Bei jeder steigenden Flanke des digitalen Eingangssignals wechselt der Ausgang seinen Zustand (toggle).

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_I1** (Typ: logiBUS::io::DI::logiBUS_IXA)
  - Parameter: QI = TRUE, Input = Input_I1
  - Zweck: Liest den physikalischen digitalen Eingang „Input_I1“ und stellt das Signal als Adapterausgang IN bereit.
- **DigitalOutput_Q1** (Typ: logiBUS::io::DQ::logiBUS_QXA)
  - Parameter: QI = TRUE, Output = Output_Q1
  - Zweck: Gibt das an seinem Adaptereingang OUT anliegende Signal an den physikalischen digitalen Ausgang „Output_Q1“ weiter.
- **AX_R_TRIG** (Typ: adapter::events::unidirectional::AX_R_TRIG)
  - Ereignisse: Ereignisausgang EO wird bei einer steigenden Flanke am Adaptereingang QI ausgelöst.
- **AX_T_FF** (Typ: adapter::events::unidirectional::AX_T_FF)
  - Ereignisse: Ereigniseingang CLK toggelt den internen Zustand bei jedem anliegenden Ereignis. Der Ausgang Q spiegelt den aktuellen Zustand wider.

Programmablauf und Verbindungen

Der Programmablauf ist über Adapter- und Ereignisverbindungen im Netzwerk festgelegt:

1. **Adapterverbindung** von `DigitalInput_I1.IN` zu `AX_R_TRIG.QI`:
   Das Eingangssignal wird auf den Adaptereingang QI des Flankendetektors gegeben.
2. **Ereignisverbindung** von `AX_R_TRIG.EO` zu `AX_T_FF.CLK`:
   Bei einer steigenden Flanke am Eingang erzeugt AX_R_TRIG ein Ereignis, das den Toggle-Flipflop taktet.
3. **Adapterverbindung** von `AX_T_FF.Q` zu `DigitalOutput_Q1.OUT`:
   Der aktuelle Ausgangszustand des Toggle-Flipflops wird auf den Ausgangsadapter übertragen.

Ablauf:

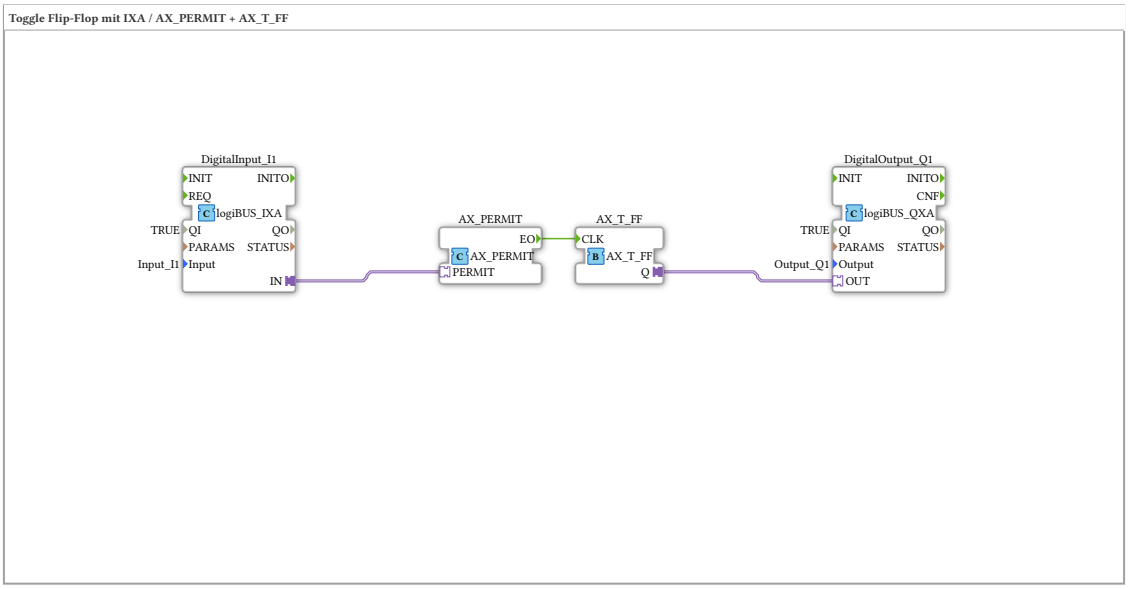
- Der digitale Eingang (z. B. ein Taster) liefert ein Signal.
- Jede steigende Flanke (Signalwechsel von 0⟶1) am Eingang wird von `AX_R_TRIG` erkannt.
- Daraufhin wird ein Ereignis an `AX_T_FF` gesendet, dessen Ausgang seinen Zustand umkehrt (toggle).

- Der geänderte Zustand erscheint am digitalen Ausgang.

Zusammenfassung

Die Übung demonstriert die Kombination von Adapter-Funktionsbausteinen zur Realisierung eines Toggle-Flipflops. Sie verbindet einen physikalischen Eingang über einen Flankendetektor mit einem Toggle-Element und gibt das Ergebnis an einen physikalischen Ausgang weiter. Dies ist eine typische Grundstruktur für das Umschalten eines Ausgangs mit einem einzelnen Taster in der Automatisierungstechnik.

2.1.2.69 Uebung_004b6_AX: Toggle Flip-Flop mit IXA / AX_PERMIT + AX_T_FF



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung demonstriert die Verwendung eines Toggle-Flip-Flops (T-FF) in Kombination mit einem Freigabe-Mechanismus. Ein digitaler Eingang (logiBUS_IXA) wird über einen **AX_PERMIT**-Adapter an den Takteingang eines **AX_T_FF**-Adapters weitergeleitet. Der Ausgang des T-FF steuert einen digitalen Ausgang (logiBUS_QXA). Dadurch kann der Zustand des Ausgangs bei jeder steigenden Flanke am Eingang umgeschaltet werden – jedoch nur, wenn der Eingang vorher vom **AX_PERMIT** freigegeben wurde.

Verwendete Funktionsbausteine (FBs)

FB-Instanz	Typ	Parameter
`DigitalInput_I1`	`logiBUS::io::DI::logiBUS_IXA`	`QI = TRUE, Input = "Input_I1"`
`DigitalOutput_Q1`	`logiBUS::io::DQ::logiBUS_QXA`	`QI = TRUE, Output = "Output_Q1"`
`AX_PERMIT`	`adapter::events::unidirectional::AX_PERMIT`	(keine Parameter gesetzt)
`AX_T_FF`	`adapter::events::unidirectional::AX_T_FF`	(keine Parameter gesetzt)

Kurzbeschreibung der Adapter

- **logiBUS_IXA**: Liest einen digitalen Eingang der logiBUS-Hardware ein.
- **logiBUS_QXA**: Schaltet einen digitalen Ausgang der logiBUS-Hardware.
- **AX_PERMIT**: Ein Adapter, der ein eingehendes Ereignis nur dann weitergibt, wenn der angeschlossene Adapter-Eingang (hier der Eingangswert) aktiv ist.
- **AX_T_FF**: Ein Toggle-Flip-Flop-Adapter. Bei jedem Ereignis am Takteingang (CLK) wird der interne Zustand umgeschaltet und über den Ausgang (Q) bereitgestellt.

Programmablauf und Verbindungen

Das Diagramm zeigt folgende Verbindungen (aus der SubAppNetwork-Konfiguration):

1. **Adapterverbindung**:
   - `DigitalInput_I1.IN` (Ausgang des Eingangsbausteins) ⟶ `AX_PERMIT.PERMIT`
     (Der digitale Eingangswert aktiviert die Freigabe des AX_PERMIT-Adapters.)
2. **Ereignisverbindung**:
   - `AX_PERMIT.EO` (Ereignisausgang des Freigabeadapters) ⟶ `AX_T_FF.CLK`
     (Nur wenn die Freigabe aktiv ist, wird ein Ereignis an den Takteingang des T-FF gesendet.)
3. **Adapterverbindung**:
   - `AX_T_FF.Q` (Zustandsausgang des T-FF) ⟶ `DigitalOutput_Q1.OUT`
     (Der aktuelle Flip-Flop-Zustand wird an den digitalen Ausgang weitergegeben.)

Funktionsweise im Detail:

- Der Eingang `Input_I1` wird ständig von `DigitalInput_I1` gelesen.
- Der gelesene Wert (TRUE/FALSE) wird als Freigabesignal an `AX_PERMIT.PERMIT` übergeben.
- Ein Ereignis (z. B. eine steigende Flanke am Eingang) tritt nicht explizit auf – die Ereignissteuerung erfolgt über die Laufzeitumgebung. Der **AX_PERMIT**-Adapter gibt ein intern generiertes Ereignis nur weiter, wenn der Freigabeeingang `TRUE` ist.
- Dieses Ereignis gelangt an den Takteingang des **AX_T_FF**, der daraufhin seinen Ausgangszustand umschaltet.
- Der neue Zustand wird am Ausgang `Output_Q1` ausgegeben.

Die Kombination erlaubt es, mit einem einzigen digitalen Eingang einen Ausgang wechselweise ein- und auszuschalten – jedoch nur, solange der Eingang aktiv ist (Freigabe). Sobald der Eingang auf FALSE geht, werden keine weiteren Taktimpulse mehr durchgelassen, und der Ausgang behält seinen letzten Zustand.

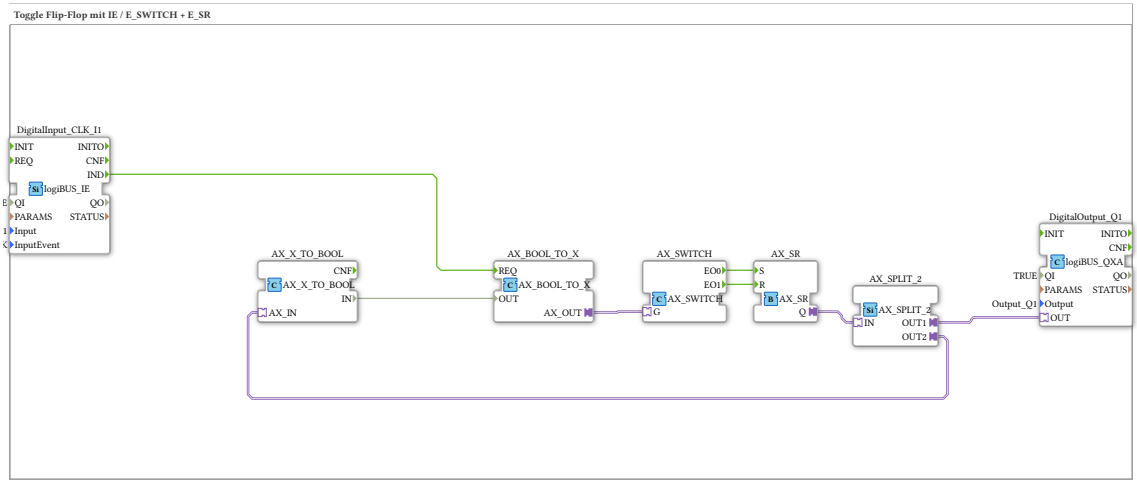
Zusammenfassung

Die Übung „Uebung_004b6_AX“ veranschaulicht die Kopplung von Hardware-Adaptern (logiBUS) mit Ereignis-/Daten-Adaptern (AX_PERMIT, AX_T_FF). Sie zeigt, wie ein Toggle-Flip-Flop realisiert werden kann, dessen Takt freigabegesteuert ist. Dieses Grundprinzip findet Anwendung in der Steuerungstechnik, z. B. zum Umschalten von Betriebsarten oder zum Entprellen von Signalen.

2.1.2.70 Uebung_004b_AX: Toggle Flip-Flop mit IE / E_SWITCH + E_SR

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_004b_AX`. Diese Übung zeigt eine alternative Implementierung eines Stromstoßschalters unter Verwendung von Daten-zu-Event-Konvertierung und Weichen.



Hinweis: Diese Lösung gilt als „nicht empfohlen“ (siehe Kommentar im Code), da sie unnötig komplex ist. Sie dient hier als Lehrbeispiel für die Bausteine `AX_SWITCH`, `AX_BOOL_TO_X` und `AX_X_TO_BOOL`.

Ziel der Übung

Verständnis der Interaktion zwischen booleschen Daten und Event-Fluss-Steuerung.

Beschreibung und Komponenten

Die Subapplikation `Uebung_004b_AX.SUB` baut einen Toggle-Mechanismus diskret auf.

Funktionsbausteine (FBs)

- **`DigitalInput_CLK_I1`**: Liefert ein Event bei Klick.
- **`AX_BOOL_TO_X`**: Wandelt einen booleschen Wert in ein Adapter-Signal (Daten + Event) um. Hier genutzt, um den aktuellen Zustand des Flip-Flops in ein Steuersignal für den Switch zu wandeln.
- **`AX_SWITCH`**: Eine Weiche. Je nach Wert am Eingang `G` leitet sie ein Event entweder an `EO0` oder `EO1`.
- **`E_SR`**: Set/Reset Flip-Flop (Ereignisbasiert).
- **`AX_SPLIT_2`**: Verteilt den Ausgang des Flip-Flops (einmal zur Lampe, einmal zur Rückkopplung).
- **`AX_X_TO_BOOL`**: Extrahiert den booleschen Zustand aus dem Adapter-Signal für die Rückkopplung.

Funktionsweise

Der Grundgedanke ist:

1. Ein Klick-Event kommt an.
2. Wohin soll es gehen? -> Zum „Einschalten“ (`S`) oder zum „Ausschalten“ (`R`)?
3. Das entscheidet der `AX_SWITCH` basierend auf dem *aktuellen* Zustand.
   - Ist die Lampe aus (`G=0`), geht das Event zu `EO0` -> `E_SR.S` (Setzen).
   - Ist die Lampe an (`G=1`), geht das Event zu `EO1` -> `E_SR.R` (Rücksetzen).

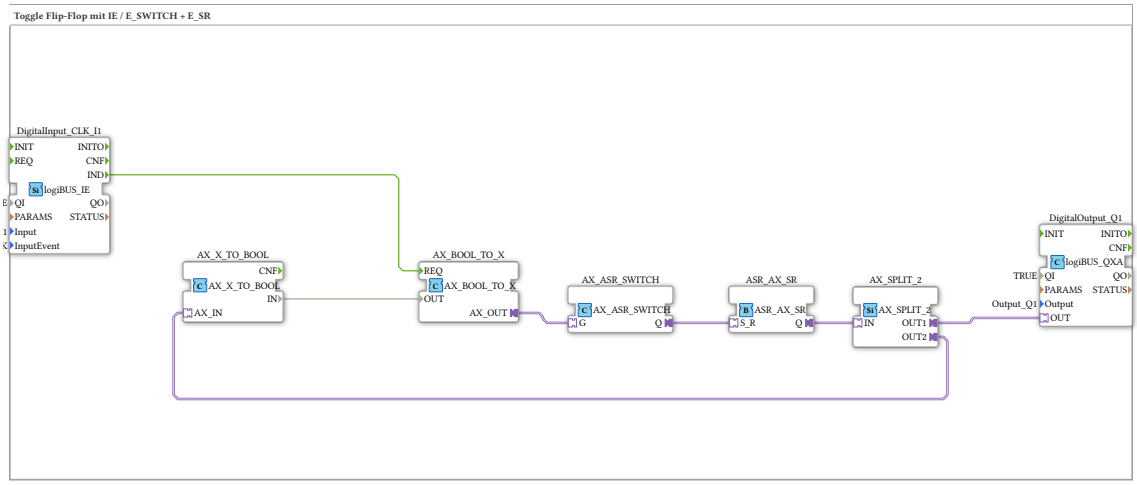
Diese Rückkopplungsschleife (Feedback Loop) macht aus dem SR-Flip-Flop effektiv ein Toggle-Flip-Flop.

Bewertung

Warum ist das „schlecht“?

- Hoher Baustein-Aufwand für eine simple Funktion.
- Rückkopplungsschleifen können in ereignisbasierten Systemen zu „Race Conditions“ oder unendlichen Loops führen, wenn man nicht aufpasst (hier durch die Trennung von Event und Datenpfad zwar funktional, aber schwer lesbar).
- Ein einfaches `AX_T_FF` (wie in Übung 004a) erledigt dasselbe in einem Baustein.

2.1.2.71 Uebung_004b_AX_ASR: Toggle Flip-Flop mit IE / E_SWITCH + E_SR



• – ○ • – ○ • – ○ •

Einleitung

Die Übung **Uebung_004b_AX_ASR** implementiert eine Toggle-Flip-Flop-Logik (Umschalter), jedoch unter Verwendung spezieller Adapter-Bausteine anstelle der klassischen booleschen Logikbausteine. Ziel ist es, den Status eines digitalen Ausgangs (Q1) bei jedem Klick auf einen Taster (I1) zu wechseln (Ein/Aus).

Diese Übung dient primär Demonstrationszwecken, um die Funktionsweise und Verkettung von Adapter-Events und -Daten zu zeigen. Wie im Quellcode vermerkt, ist diese Umsetzung für einfache Schaltaufgaben aufgrund der hohen Komplexität und Anzahl der Bausteine in der Praxis **nicht empfohlen**.

Verwendete Funktionsbausteine (FBs)

In dieser Sub-Application werden verschiedene Bausteine aus der `logiBUS` und der `adapter` Bibliothek verwendet, um die Logik abzubilden.

LogiBUS IO Bausteine

- **DigitalInput_CLK_I1** (`logiBUS::io::DI::logiBUS_IE`)
  - Dient als Eingangsereignis.
  - **Parameter**: `Input` = `Input_I1` (Logischer Eingang 1), `InputEvent` = `BUTTON_SINGLE_CLICK` (Reagiert auf einfachen Klick).
  - **Funktion**: Liefert ein Event-Signal, wenn der Taster gedrückt wird.
- **DigitalOutput_Q1** (`logiBUS::io::DQ::logiBUS_QXA`)
  - Dient als Ausgangsschnittstelle.
  - **Parameter**: `Output` = `Output_Q1`.
  - **Funktion**: Steuert den physischen Ausgang basierend auf dem Adapter-Signal.

Adapter-Logik Bausteine

Diese Bausteine verarbeiten Signale über Adapter-Verbindungen (`AX_...`), welche Daten und Events kapseln.

- **AX_SR** (`adapter::events::unidirectional::AX_SR`)
  - **Typ**: Set/Reset-Flip-Flop (Adapter-Variante).
  - **Funktion**: Speichert den Zustand (Ein/Aus). Er wird durch Events an den Eingängen `S` (Set) und `R` (Reset) gesteuert und gibt den Status über den Adapter-Ausgang `Q` aus.
- **AX_SWITCH** (`adapter::events::unidirectional::AX_SWITCH`)
  - **Typ**: Weiche/Schalter.
  - **Funktion**: Leitet eingehende Signale basierend auf dem Zustand an verschiedene Ausgänge (`EO0`, `EO1`) weiter. Hier genutzt, um zwischen Setzen und Rücksetzen des Flip-Flops zu wechseln.
- **AX_SPLIT_2** (`adapter::events::unidirectional::AX_SPLIT_2`)
  - **Typ**: Signal-Splitter.
  - **Funktion**: Teilt den Adapter-Ausgang des Flip-Flops auf zwei Pfade auf: einen für den physischen Ausgang und einen für die Rückkopplung (Feedback).
- **AX_BOOL_TO_X** (`adapter::conversion::unidirectional::AX_BOOL_TO_X`)
  - **Typ**: Konverter.
  - **Funktion**: Wandelt ein klassisches Event- und Datensignal in ein Adapter-Signal um, um den `AX_SWITCH` anzusteuern.
- **AX_X_TO_BOOL** (`adapter::conversion::unidirectional::AX_X_TO_BOOL`)
  - **Typ**: Konverter.
  - **Funktion**: Wandelt ein Adapter-Signal zurück in klassische Daten, um den aktuellen Status für die Rückkopplung bereitzustellen.

Programmablauf und Verbindungen

Der Ablauf simuliert ein T-Flip-Flop durch eine Rückkopplungsschleife:

1. **Eingangssignal**: Das Event `IND` vom Taster **DigitalInput_CLK_I1** triggert den Baustein **AX_BOOL_TO_X**.
2. **Status-Erfassung**: Der aktuelle Status des Systems wird über eine Rückkopplung (Feedback Loop) ermittelt. Der Ausgang des Flip-Flops (**AX_SR**) wird über **AX_SPLIT_2** und **AX_X_TO_BOOL** zurückgeführt und in **AX_BOOL_TO_X** eingespeist.
3. **Schaltlogik**:
   - Der **AX_SWITCH** erhält das Signal.

- Abhängig vom aktuellen Zustand (Rückkopplung) wird entweder der Ausgang `E00` (verbunden mit `AX_SR.S` -> Setzen) oder `E01` (verbunden mit `AX_SR.R` -> Rücksetzen) aktiviert.

4. **Speicherung**: Der **AX_SR** Baustein ändert seinen Zustand entsprechend (Toggeln).
5. **Ausgabe**: Der neue Zustand steht am Adapter-Ausgang `Q` von **AX_SR** bereit.
6. **Verteilung**:
   - Über **AX_SPLIT_2** geht das Signal an den **DigitalOutput_Q1**, wodurch die Lampe/der Aktor geschaltet wird.
   - Gleichzeitig wird das Signal für den nächsten Klick wieder in die Rückkopplungsschleife geleitet.

Hinweis zur Komplexität: Im Netzwerk-Diagramm ist ein Kommentar hinterlegt: *„nicht empfohlen !!! viel zu viel Bausteine“*. Dies unterstreicht, dass diese Lösung für eine einfache Toggle-Funktion in einer produktiven Umgebung übermäßig komplex ist (Over-Engineering). Ein einfacher `E_T_FF` (Toggle Flip-Flop) oder eine Kombination aus `E_SWITCH` und `E_SR` ohne Adapter-Kapselung wäre effizienter.

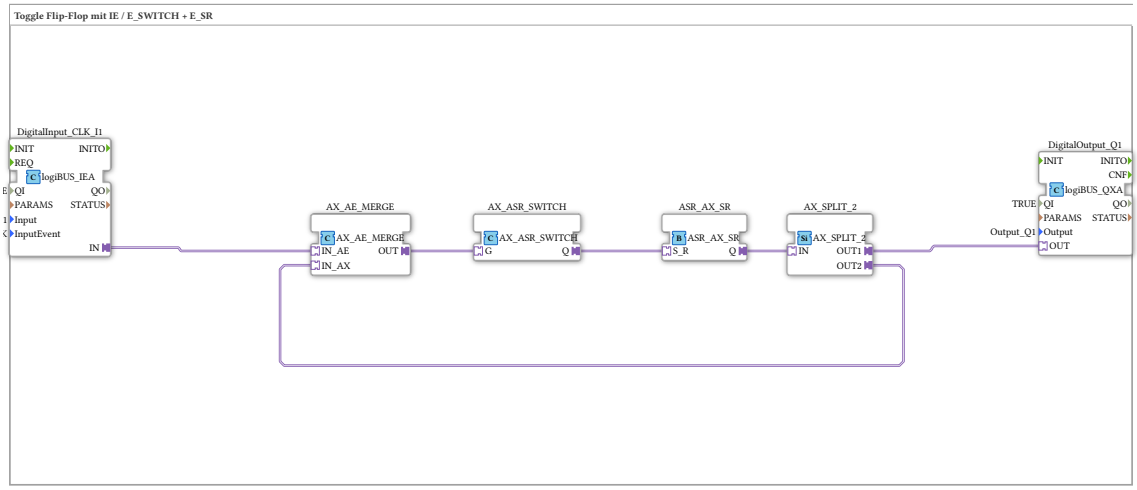
Zusammenfassung

Die Übung `Uebung_004b_AX_ASR` demonstriert die Realisierung eines Toggle-Flip-Flops unter ausschließlicher Verwendung von Adapter-Bausteinen (`AX_`) und Konvertern. Sie zeigt anschaulich, wie Adapter-Verbindungen gesplittet und konvertiert werden können, dient aber gleichzeitig als Negativbeispiel für die Effizienz bei simplen Logikaufgaben. Das Lernziel liegt im Verständnis der Adapter-Technologie innerhalb der 4diac IDE.

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

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

2.1.2.72 Uebung_004b_AX_ASR_X: Toggle Flip-Flop mit IE / E_SWITCH + E_SR



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung implementiert ein **Toggle-Flip-Flop** (Umschalter), jedoch unter Verwendung eines sehr speziellen Ansatzes mittels Adapter-Bausteinen (`AX`). Ziel der Schaltung ist es, bei einem Eingangssignal (Tastendruck) den Ausgangszustand zu wechseln (An -> Aus -> An).

Besonderheit dieser Übung ist der Hinweis im Quellcode, dass dieser Lösungsweg aufgrund der hohen Anzahl an Bausteinen für diese simple Aufgabe **nicht empfohlen** ist. Sie dient primär dem Verständnis von Adapter-Verbindungen, Signal-Splitting und Rückkopplungsschleifen in 4diac.

Verwendete Funktionsbausteine (FBs)

In dieser Sub-Application kommen verschiedene Bausteine aus der `logiBUS` Bibliothek für die IO-Anbindung sowie Bausteine aus der `adapter` Bibliothek für die logische Verarbeitung zum Einsatz.

Haupt-Bausteine

DigitalInput_CLK_I1

- **Typ**: `logiBUS::io::DI::logiBUS_IE`
- **Beschreibung**: Dieser Baustein erfasst das Eingangssignal (Taster).
- **Parameter**:
  - `QI` = `TRUE`
  - `Input` = `Input_I1`
  - `InputEvent` = `BUTTON_SINGLE_CLICK`
- **Funktion**: Sendet ein Event (`IND`), wenn die Taste einmal gedrückt wird.

DigitalOutput_Q1

- **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
- **Beschreibung**: Steuert den physischen Ausgang an.
- **Parameter**:
  - `QI` = `TRUE`
  - `Output` = `Output_Q1`
- **Funktion**: Übernimmt den Status vom Adapter-Netzwerk und schaltet den Ausgang entsprechend.

AX_SR

- **Typ**: `adapter::events::unidirectional::AX_SR`
- **Beschreibung**: Ein Speicher-Baustein (Set/Reset) auf Adapter-Basis.
- **Funktion**: Speichert den aktuellen Zustand (TRUE oder FALSE). Er wird über die Eingänge `S` (Setzen) und `R` (Rücksetzen) gesteuert.

AX_SWITCH

- **Typ**: `adapter::events::unidirectional::AX_SWITCH`
- **Beschreibung**: Dient als Weiche für Events/Adapter-Signale.
- **Funktion**: Leitet das eingehende Signal basierend auf dem Status am Eingang `G` entweder auf `EO0` oder `EO1` weiter.

AX_SPLIT_2

- **Typ**: `adapter::events::unidirectional::AX_SPLIT_2`
- **Beschreibung**: Ein Splitter-Baustein.
- **Funktion**: Teilt ein eingehendes Adapter-Signal (`IN`) auf zwei Ausgänge (`OUT1`, `OUT2`) auf. Dies wird hier benötigt, um den Ausgangszustand gleichzeitig an den physischen Ausgang zu senden und als Rückkopplung für die Logik zu nutzen.

AX_BOOL_TO_X & AX_X_TO_BOOL

- **Typ**: `adapter::conversion::unidirectional::...`
- **Beschreibung**: Konvertierungsbausteine.
- **Funktion**: Dienen der Umwandlung zwischen klassischen Datentypen und Adapter-Strukturen, um die Rückkopplungsschleife zu schließen.

Programmablauf und Verbindungen

Die Logik dieser Übung basiert auf einer Rückkopplung des aktuellen Zustands, um zu entscheiden, ob beim nächsten Tastendruck eingeschaltet (Set) oder ausgeschaltet (Reset) werden soll.

1. **Eingangssignal**: Ein Klick auf `DigitalInput_CLK_I1` löst ein Event aus (`IND`), welches den Konverter `AX_BOOL_TO_X` aktiviert (`REQ`).
2. **Entscheidungslogik (Weiche)**: Das Signal gelangt zum `AX_SWITCH` (Eingang `G`).
3. **Zustandsänderung**:
   - Der `AX_SWITCH` ist mit dem `AX_SR` (Speicher) verbunden.
   - Über `EO0` wird der Speicher gesetzt (`S`).
   - Über `EO1` wird der Speicher zurückgesetzt (`R`).
   - Welcher Weg gewählt wird, hängt vom aktuellen Zustand der Rückkopplung ab.
4. **Ausgabe und Rückkopplung**:
   - Der Ausgang des Speichers `AX_SR` geht in den Splitter `AX_SPLIT_2`.
   - **Zweig 1 (`OUT1`)**: Geht direkt an den `DigitalOutput_Q1`, um die Lampe zu schalten.

- **Zweig 2 (`OUT2`)**: Wird zurückgeführt. Er läuft über `AX_X_TO_BOOL` (Konvertierung) zurück zu `AX_BOOL_TO_X`.

5. **Der Zyklus**: Durch diese Rückführung (Feedback Loop) „weiß“ das System beim nächsten Klick, in welchem Zustand es sich befindet, und der `AX_SWITCH` schaltet entsprechend in den entgegengesetzten Zustand.

Zusammenfassung

Die Übung **Uebung_004b_AX_ASR_X** demonstriert die Erstellung eines Toggle-Flip-Flops unter ausschließlicher Verwendung von Adapter-Event-Bausteinen und Konvertern.

Obwohl die Funktionalität (Taster drücken -> Licht an, Taster drücken -> Licht aus) gegeben ist, zeigt der interne Kommentar („nicht empfohlen !!! viel zu viel Bausteine“), dass dies eine akademische Übung ist. Sie verdeutlicht, wie man komplexe Adapter-Netzwerke mit Rückkopplungen und Signalweichen (`SWITCH` und `SPLIT`) aufbaut, stellt jedoch keine effiziente Lösung für eine einfache Stromstoßschaltung dar.

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

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

2.1.2.73 Uebung_004c1_AX: Toggle Flip-Flop mit IE mit BUTTON_DOUBLE_CLICK



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_004c1_AX`. Ab hier widmen wir uns den erweiterten Fähigkeiten des `logiBUS_IE` Bausteins, der komplexe Taster-Muster erkennen kann.

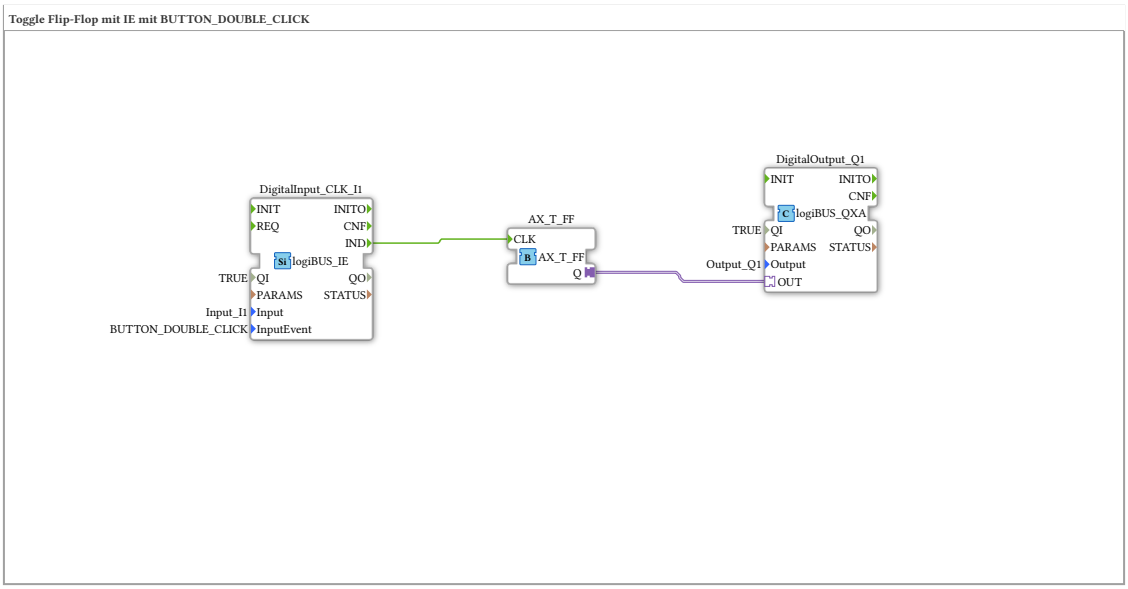
Ziel der Übung

Nutzung des Ereignisses `BUTTON_DOUBLE_CLICK`.

Beschreibung und Komponenten

Die Subapplikation `Uebung_004c1_AX.SUB` schaltet eine Lampe nur bei einem Doppelklick um.

Funktionsbausteine (FBs)



- **`DigitalInput_CLK_I1`**: Konfiguriert mit `InputEvent = BUTTON_DOUBLE_CLICK`.

Funktionsweise

Der Baustein überwacht den Eingang `I1`.

1. Drückt man einmal: Nichts passiert am Ausgang `IND`.
2. Drückt man zweimal kurz hintereinander (innerhalb einer definierten Zeitspanne): Der Baustein feuert *ein* `IND` Event.
3. Das Flip-Flop toggelt.

Anwendungsbeispiel

Vermeidung von Fehlbedienung: Kritische Funktionen (z.B. „Alle Löschen“) auf einen Doppelklick legen, damit sie nicht versehentlich ausgelöst werden.

2.1.2.74 Uebung_004c2_AX: Toggle Flip-Flop mit IE mit BUTTON_LONG_PRESS_START



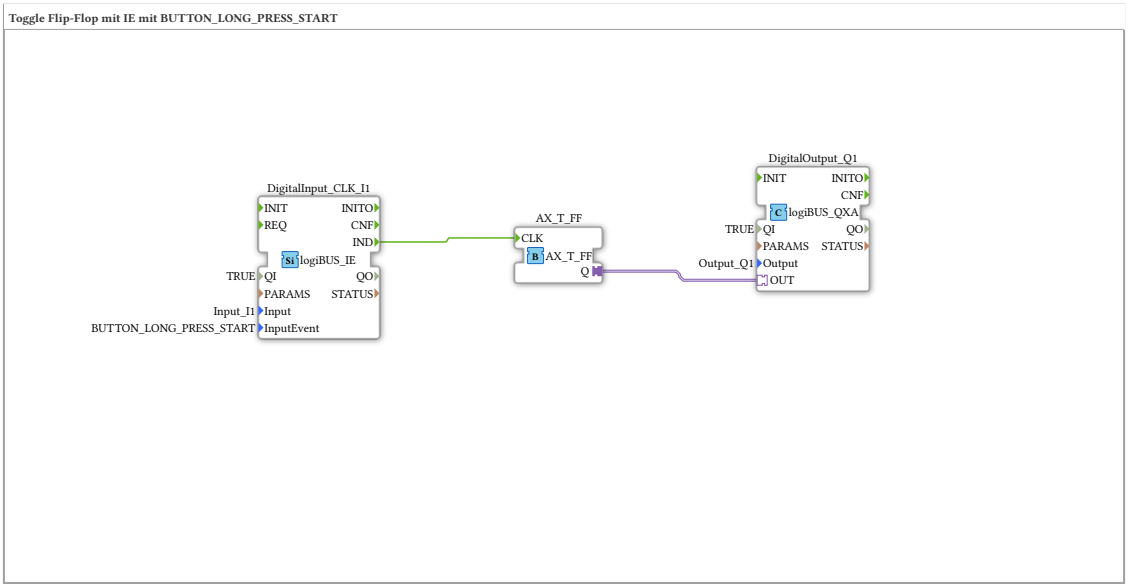
Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_004c2_AX`.

Ziel der Übung

Nutzung des Ereignisses `BUTTON_LONG_PRESS_START`.

Funktionsweise

Der Baustein `DigitalInput_CLK_I1` in `Uebung_004c2_AX.SUB` ist auf `BUTTON_LONG_PRESS_START` konfiguriert.



Das Event `IND` wird gefeuert, sobald der Taster eine bestimmte Zeit lang gedrückt *gehalten wurde* (z.B. > 500ms). Es feuert genau in dem Moment, in dem die Zeit abgelaufen ist, auch wenn der Taster noch weiter gedrückt bleibt.

Anwendungsbeispiel

Licht dimmen: Ein kurzer Klick schaltet Ein/Aus (siehe Übung 004a). Ein langer Druck (erkannt durch `LONG_PRESS_START`) startet den Dimm-Vorgang.

2.1.2.75 Uebung_004c3_AX: Toggle Flip-Flop mit IE mit BUTTON_LONG_PRESS_UP

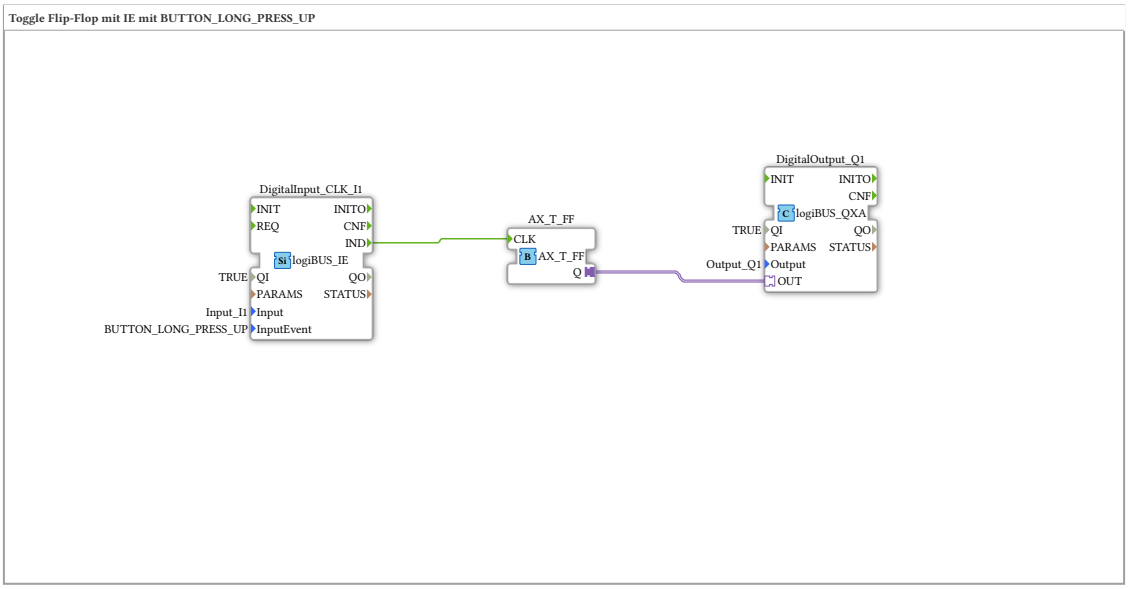
Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_004c3_AX`.

Ziel der Übung

Nutzung des Ereignisses `BUTTON_LONG_PRESS_UP`.

Funktionsweise

Der Baustein `DigitalInput_CLK_I1` in `Uebung_004c3_AX.SUB` ist auf `BUTTON_LONG_PRESS_UP` konfiguriert.



Das Event wird gefeuert, wenn der Taster losgelassen wird, *nachdem* er als „lang gedrückt“ erkannt wurde. Ein kurzes Tippen löst dieses Event nicht aus (das wäre `SINGLE_CLICK`).

Anwendungsbeispiel

Dimm-Vorgang beenden: Wenn der Benutzer den Taster loslässt, soll das Dimmen stoppen und der aktuelle Helligkeitswert gespeichert werden.

2.1.2.76 Uebung_004c4_AX: Toggle Flip-Flop mit IE mit BUTTON_LONG_PRESS_HOLD

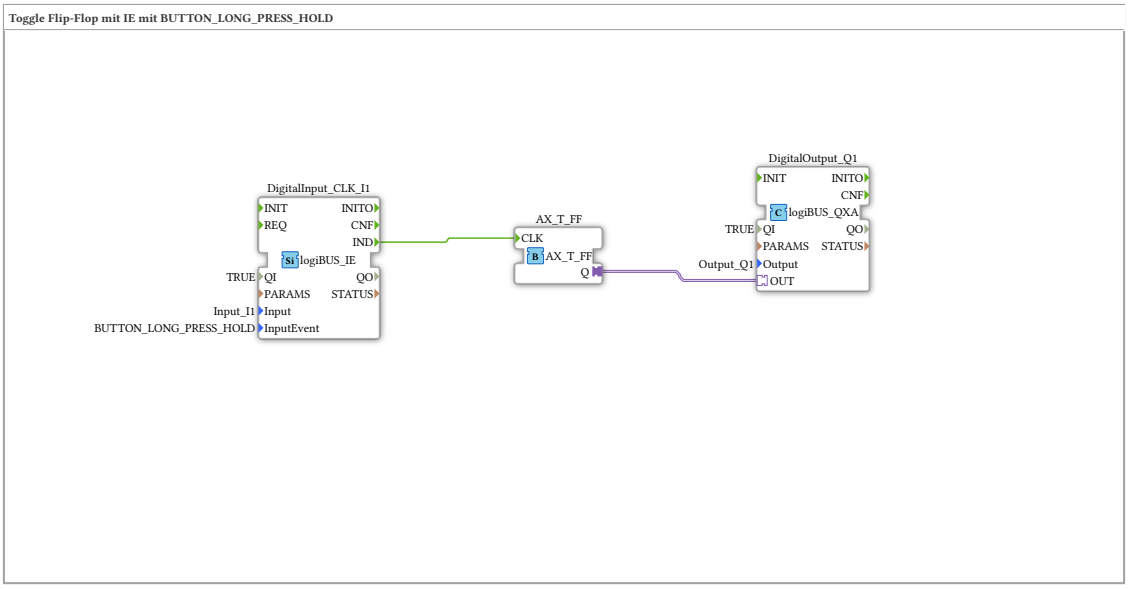
Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_004c4_AX`.

Ziel der Übung

Nutzung des Ereignisses `BUTTON_LONG_PRESS_HOLD`.

Funktionsweise

Der Baustein `DigitalInput_CLK_I1` in `Uebung_004c4_AX.SUB` ist auf `BUTTON_LONG_PRESS_HOLD` konfiguriert.



Dieses Event wird *periodisch* gefeuert, solange der Taster nach Erkennung des langen Drucks weiterhin gehalten wird.

Anwendungsbeispiel

Lautstärke ändern / Scrollen: Solange man die Taste gedrückt hält, wird die Lautstärke schrittweise erhöht oder durch ein Menü gescrollt. Das Toggle-FF würde hier sehr schnell an und aus gehen (Flackern), was zeigt, dass dieses Event eher für Inkrement-Funktionen als für Toggles gedacht ist.

2.1.2.77 Uebung_004c5_AX: Toggle Flip-Flop mit IE mit BUTTON_PRESS_END

NotebookLM

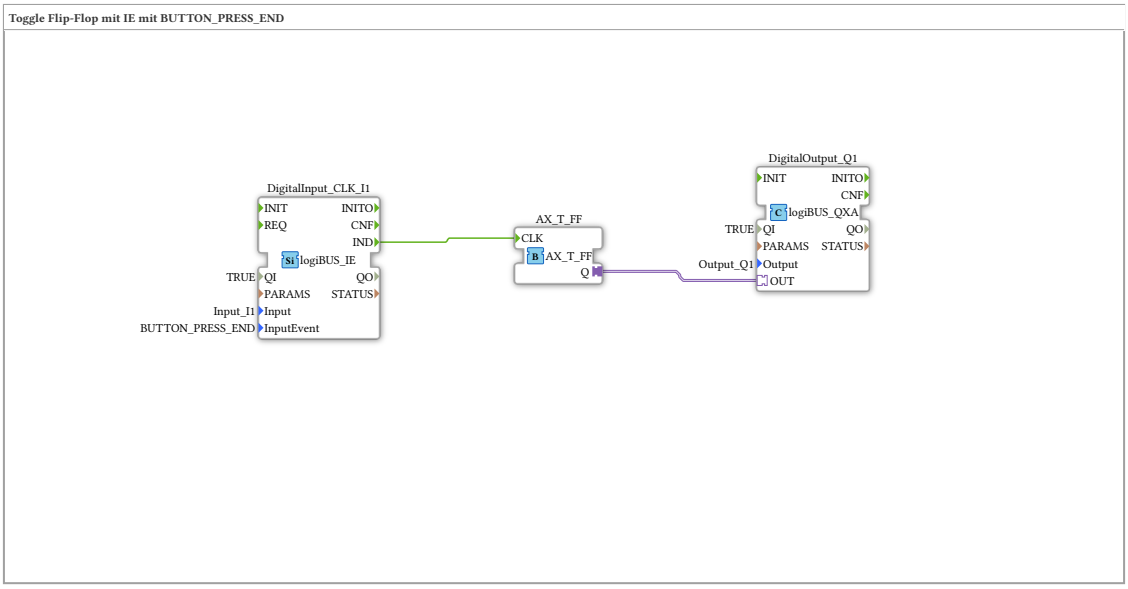
Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_004c5_AX`.

Ziel der Übung

Nutzung des Ereignisses `BUTTON_PRESS_END`.

Funktionsweise

Der Baustein `DigitalInput_CLK_I1` in `Uebung_004c5_AX.SUB` ist auf `BUTTON_PRESS_END` konfiguriert.



Dieses Event feuert *immer*, wenn der Taster losgelassen wird, egal ob kurz oder lang gedrückt wurde.

Anwendungsbeispiel

Totmannschalter: Eine Funktion ist aktiv, solange gedrückt wird (Start beim Pegel-Wechsel auf HIGH), und muss sicher stoppen, wenn losgelassen wird (`PRESS_END`).

2.1.2.78 Uebung_004c6_AX: Toggle Flip-Flop mit IE mit BUTTON_MULTIPLE_CLICK (3-fach Klick)



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_004c6_AX`. Hier wird der erweiterte `logiBUS_IE2` Baustein genutzt, der Argumente akzeptiert.

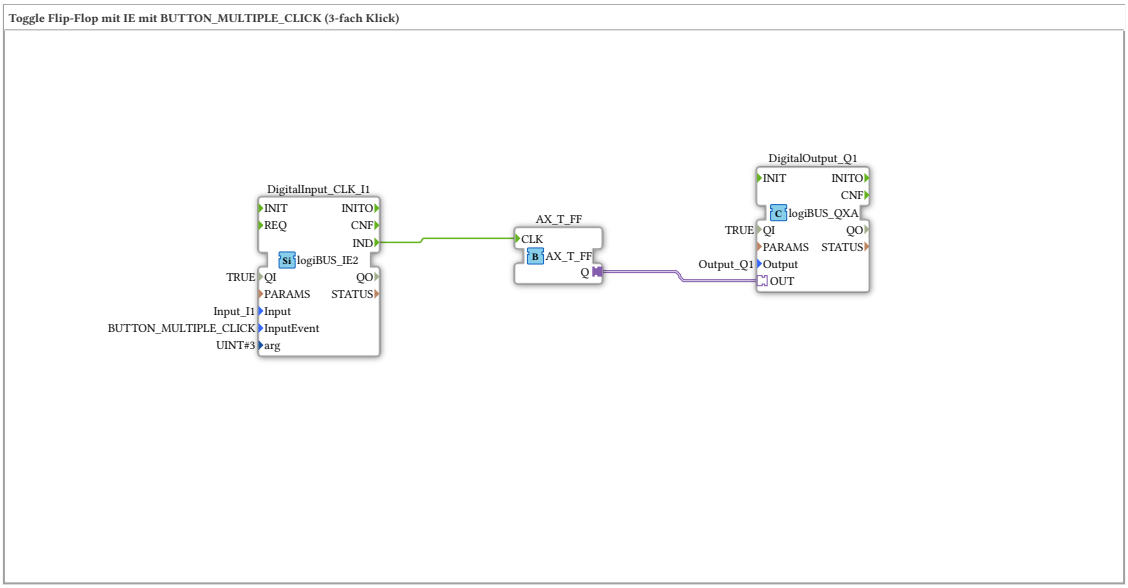
Ziel der Übung

Konfiguration eines n-fach Klicks.

Beschreibung und Komponenten

Die Subapplikation `Uebung_004c6_AX.SUB` nutzt `logiBUS_IE2` mit `InputEvent = BUTTON_MULTIPLE_CLICK` und `arg = 3`.

Funktionsbausteine (FBs)



- **`DigitalInput_CLK_I1`**: Typ `logiBUS_IE2`. Dieser Typ hat den zusätzlichen Eingang `arg`.

Funktionsweise

Das Event feuert nur, wenn der Benutzer exakt dreimal kurz hintereinander klickt (Triple-Click).

Anwendungsbeispiel

Versteckte Service-Menüs: Zugriff auf Experten-Einstellungen, die ein normaler Nutzer nicht versehentlich aktivieren soll.

2.1.2.79 Uebung_004c7_AX: Toggle Flip-Flop mit IE mit BUTTON_PRESS_END

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_004c7_AX`. Auch hier wird `logiBUS_IE2` genutzt, um die Zeitdauer für einen „langen Druck“ anzupassen.

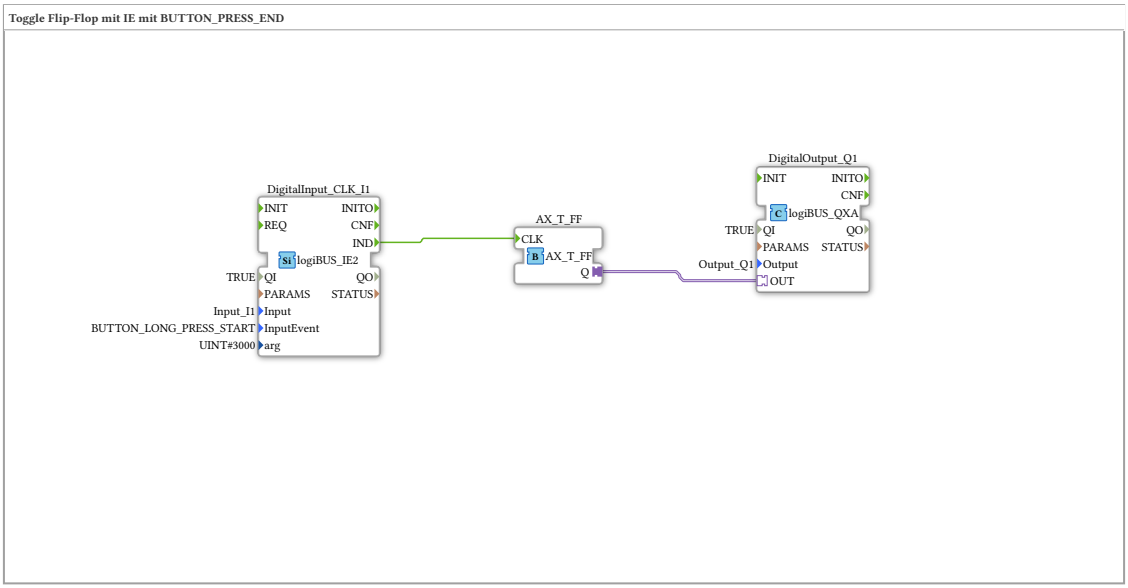
Ziel der Übung

Definition einer spezifischen Haltezeit.

Beschreibung und Komponenten

Die Subapplikation `Uebung_004c7_AX.SUB` nutzt `logiBUS_IE2` mit `InputEvent = BUTTON_LONG_PRESS_START` und `arg = 3000` ms.

Funktionsweise



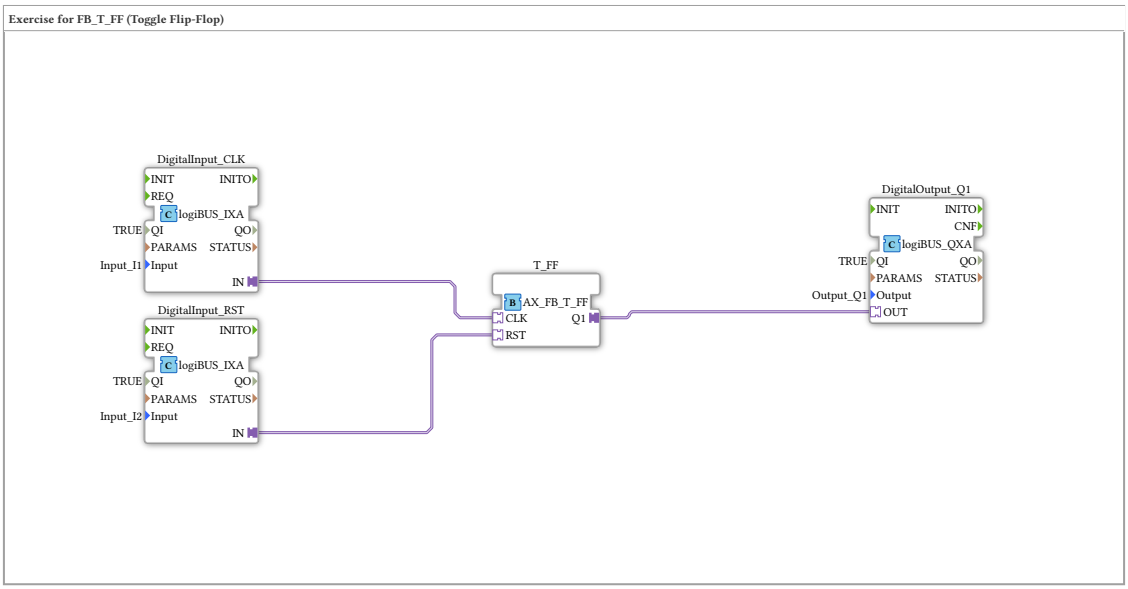
Das Event feuert erst, wenn der Taster für **3 Sekunden** (3000ms) gedrückt gehalten wird. Dies überschreibt den Standardwert (der meist bei 500ms oder 1s liegt).

Anwendungsbeispiel

Werkseinstellungen laden: Eine sehr destruktive Aktion, die extrem gut gegen versehentliches Auslösen geschützt sein muss. Der Nutzer muss bewusst lange drücken.

2.1.2.80 Uebung_004d_T_AX: Exercise for FB_T_FF (Toggle Flip-Flop)

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_004d_T_AX (Exercise for FB_T_FF (Toggle Flip-Flop)).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Exercise for FB_T_FF (Toggle Flip-Flop)**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_004d_T_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_RST**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I2
- **DigitalInput_CLK**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
- **T_FF**: Instanz des Typs adapter::bistableElements::AX_FB_T_FF.
- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1

Verbindungen und Schnittstellen

Adapterverbindungen:

- DigitalInput_RST.IN -> T_FF.RST
- DigitalInput_CLK.IN -> T_FF.CLK
- T_FF.Q1 -> DigitalOutput_Q1.OUT

Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_004d_T_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.81 Uebung_005_AX: Toggle Flip-Flop mit IXA



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_005_AX`. Diese Übung zeigt, wie man einen zustandsbasierten Eingang (`IXA`) nutzt, um ein ereignisbasiertes Flip-Flop zu steuern.

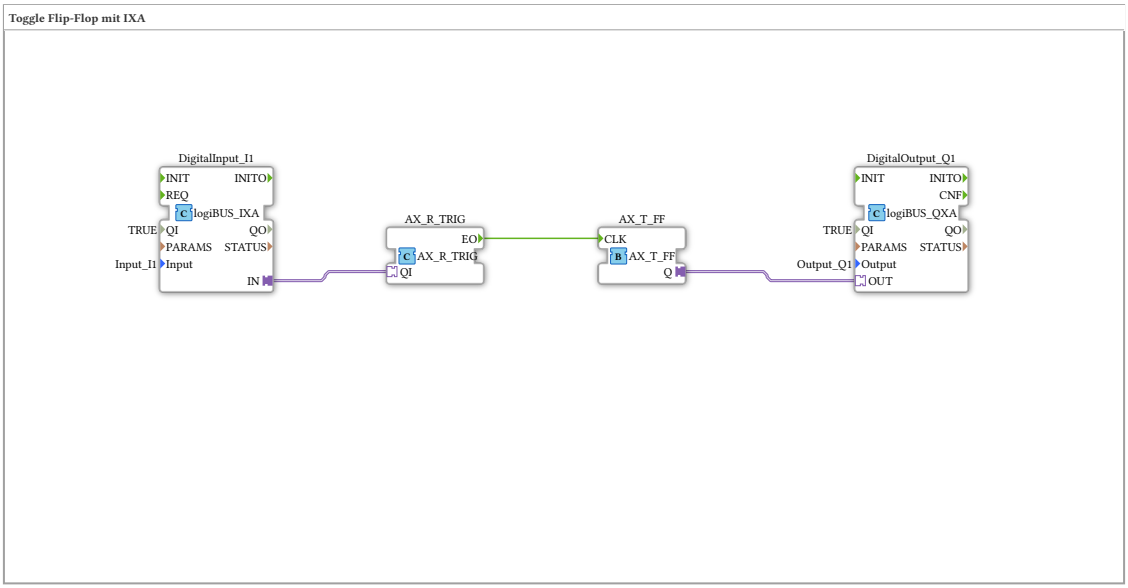
Ziel der Übung

Demonstration der Konvertierung von Daten zu Ereignissen für Steuerungszwecke.

Beschreibung und Komponenten

Die Subapplikation `Uebung_005_AX.SUB` verwendet einen Standard-Digitaleingang (`logiBUS_IXA`) anstelle eines Event-Eingangs (`logiBUS_IE`).

Funktionsbausteine (FBs)



- **`DigitalInput_I1`**: Typ `logiBUS_IXA`. Liefert kontinuierlich TRUE, wenn gedrückt.
- **`AX_SWITCH`**: Dient hier als Gatter.
- **`AX_T_FF`**: Das Toggle-Flip-Flop.

Funktionsweise

Die Schaltung nutzt die Tatsache, dass der `IXA` bei jeder Änderung auch ein Adapter-Event sendet.

1. Wenn `I1` gedrückt wird (FALSE -> TRUE), sendet der Adapter ein Event und `D1=TRUE`.
2. Der `AX_SWITCH` bekommt das Event. Da `G` (verbunden mit `I1.IN`) nun TRUE ist, leitet er das Event an `EO1` weiter.
3. `EO1` triggert das Flip-Flop -> Licht schaltet um.
4. Wenn `I1` losgelassen wird (TRUE -> FALSE), sendet der Adapter wieder ein Event, aber `D1=FALSE`.
5. Der `AX_SWITCH` leitet das Event an `EO0` (hier offen) weiter. Das Flip-Flop wird nicht getriggert.

Das Ergebnis ist eine korrekte Flankenauswertung (nur bei steigender Flanke wird geschaltet).

Bewertung

Dies ist eine valide Methode, wenn man nur `IXA` Bausteine zur Verfügung hat und keinen `IE` nutzen kann oder will. Es ist jedoch ressourcenintensiver als die Nutzung des spezialisierten `IE` Bausteins mit `SINGLE_CLICK` Event.

2.1.2.82 Uebung_006_AX: SR-Flip-Flop mit 2x IE

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_006_AX`. Hier wird das klassische RS-Glied (Speicherglied) implementiert.

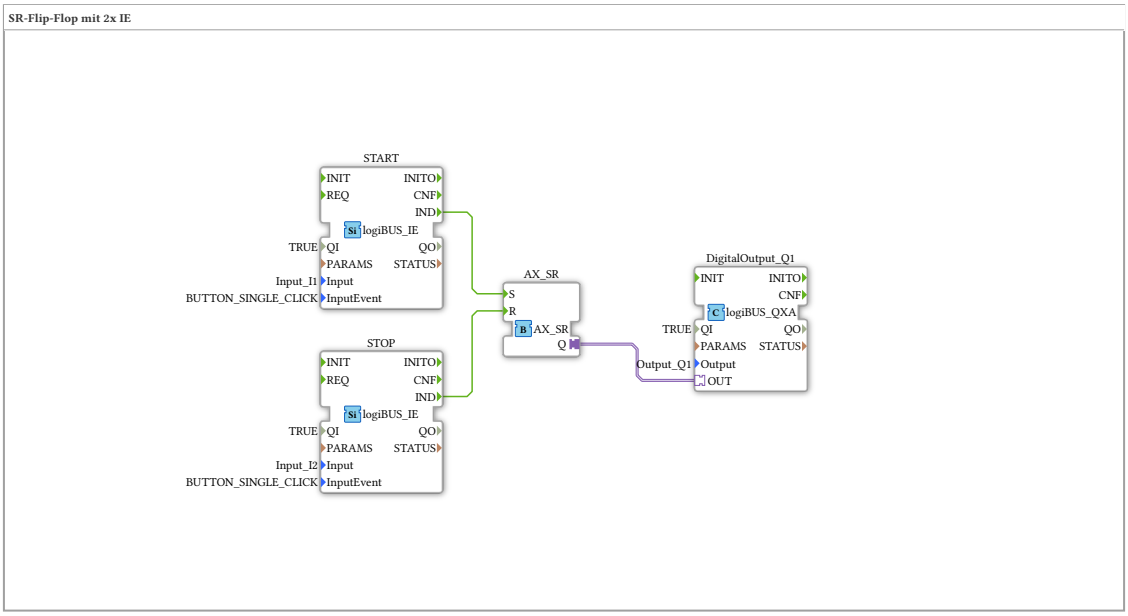
Ziel der Übung

Realisierung einer Schaltung mit getrennten Tastern für „Ein“ und „Aus“.

Beschreibung und Komponenten

Die Subapplikation `Uebung_006_AX.SUB` nutzt zwei Taster und einen `AX_SR` Baustein.

Funktionsbausteine (FBs)



- **`I1` (Set)**: Taster zum Einschalten.
- **`I2` (Reset)**: Taster zum Ausschalten.
- **`AX_SR`**: Ein SR-Flip-Flop (Set dominant, falls gleichzeitig, aber hier durch Events getrennt).

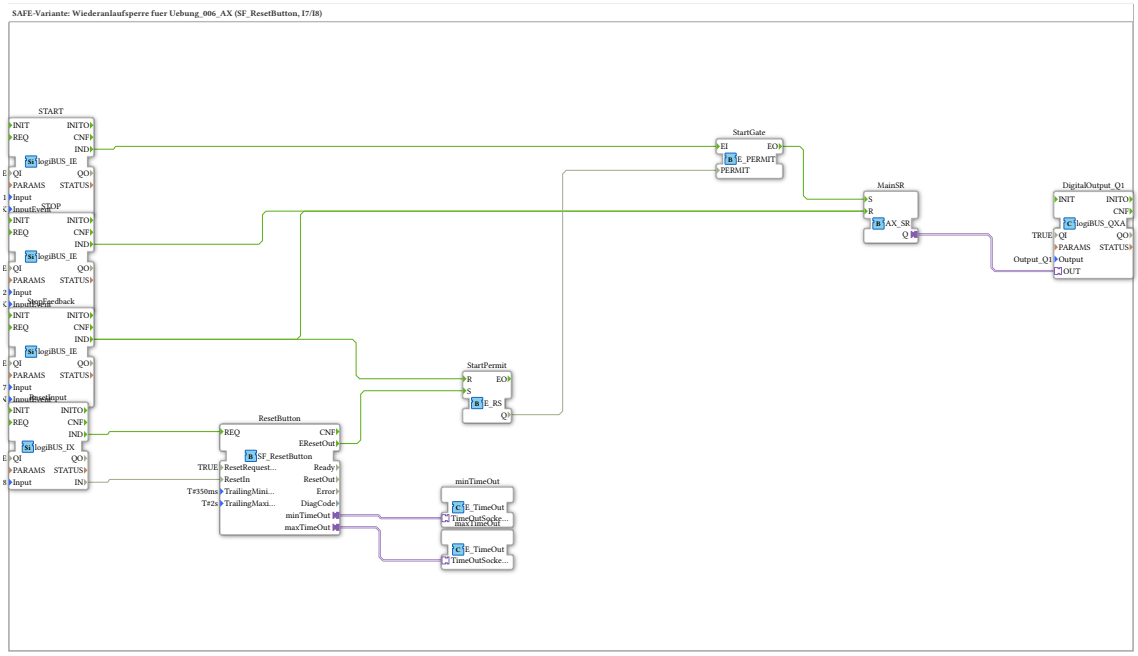
Funktionsweise

- Ein Klick auf `I1` sendet ein Event an `S` -> Ausgang `Q` wird TRUE.
- Ein Klick auf `I2` sendet ein Event an `R` -> Ausgang `Q` wird FALSE.
- Mehrmaliges Drücken von `I1` ändert nichts, wenn es schon an ist.

Anwendungsbeispiel

Maschinensteuerung: Ein grüner Taster startet den Motor, ein roter Taster stoppt ihn. Dies ist sicherer als ein Toggle-Taster, da der Bediener immer definiert „Aus“ drücken kann.

2.1.2.83 Uebung_006_SAFE_AX: SAFE-Variante mit Wiederanlaufsperre (SF_ResetButton)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung ist die SAFE-Variante von `Uebung_006_AX` (SR-Flip-Flop mit START/STOP-Tastern): STOP (`I2`) und der Meldeeingang `I7` (Rückmeldung des Hardware-Not-Halts) schalten den Ausgang softwareseitig sofort ab und sperren START über einen Freigabe-Latch (`E_RS`) und ein Start-Gatter (`E_PERMIT`). Die Sperre wird erst aufgehoben, wenn RESET (`I8`) über `SF_ResetButton` (PLCopen Safety Part 1, Kap. 6.1) eine gültige Tasten-Flanke erkennt. RESET alleine schaltet den Ausgang nicht wieder ein, dafür muss START erneut gedrückt werden.

Kaltstart: `StartPermit.Q` ist `FALSE` (Typ-Default von `BOOL`, `E_RS` hat keinen Initialwert). START bleibt also gesperrt, bis nach dem Anlauf eine gültige RESET-Quittierung erfolgt.

Verwendete Funktionsbausteine (FBs)

- **START**: Taster START
  - **Typ**: `logiBUS::io::DI::logiBUS_IE`
  - **Parameter**: `QI = TRUE`, `Input = Input_I1`, `InputEvent = BUTTON_SINGLE_CLICK`
  - **Erklärung**: Meldet einen Einzelklick auf `I1` als Event `IND`.
- **STOP**: Hand-Stopp (nicht der Hardware-Not-Halt)
  - **Typ**: `logiBUS::io::DI::logiBUS_IE`
  - **Parameter**: `QI = TRUE`, `Input = Input_I2`, `InputEvent = BUTTON_SINGLE_CLICK`
  - **Erklärung**: Meldet einen Einzelklick auf `I2` als Event `IND`.
- **StopFeedback**: Meldeeingang der Hardware-Not-Halt-Rückmeldung
  - **Typ**: `logiBUS::io::DI::logiBUS_IE`
  - **Parameter**: `QI = TRUE`, `Input = Input_I7`, `InputEvent = BUTTON_PRESS_DOWN`
  - **Erklärung**: Meldet das Drücken (`BUTTON_PRESS_DOWN`) auf `I7` als Event `IND`.
- **ResetInput**: Pegel und Event für `SF_ResetButton`
  - **Typ**: `logiBUS::io::DI::logiBUS_IX`
  - **Parameter**: `QI = TRUE`, `Input = Input_I8`
  - **Erklärung**: Liefert den Pegel von `I8` (`IN`) und das zugehörige Event `IND`.
- **ResetButton**: Quittiertaster nach PLCopen Safety
  - **Typ**: `logiBUS::safety::PLCopen::SF_ResetButton`
  - **Parameter**: `ResetRequested = TRUE`, `TrailingMinimum = T#350ms`, `TrailingMaximum = T#2s`
  - **Erklärung**: Ergänzt Reset-Eingänge um eine Trailing-Edge-Auswertung (LOW-HIGH-LOW): Nach einem gültigen Drücken und Loslassen von `I8` (Eingänge `REQ` und `ResetIn`) innerhalb von

TrailingMinimum bis TrailingMaximum wird das Event EResetOut ausgegeben. Die Zeitgrenzen werden über die Adapter minTimeOut/maxTimeOut von zwei E_TimeOut-Bausteinen überwacht.

- **minTimeOut**: Timer für die Mindestzeit von SF_ResetButton
  - **Typ**: iec61499::events::E_TimeOut
  - **Parameter**: keine
  - **Erklärung**: Über TimeOutSocket mit ResetButton.minTimeOut verbunden.
- **maxTimeOut**: Timer für die Höchstzeit von SF_ResetButton
  - **Typ**: iec61499::events::E_TimeOut
  - **Parameter**: keine
  - **Erklärung**: Über TimeOutSocket mit ResetButton.maxTimeOut verbunden.
- **StartPermit**: Freigabe-Latch (E_RS statt E_SR nur wegen kreuzungsfreier Verdrahtung)
  - **Typ**: iec61499::events::E_RS
  - **Parameter**: keine
  - **Erklärung**: S (aus ResetButton.EResetOut) gibt START frei, R (aus StopFeedback.IND) sperrt START. Der Ausgang Q speist StartGate.PERMIT.
- **StartGate**: Lässt START nur bei Freigabe durch
  - **Typ**: iec61499::events::E_PERMIT
  - **Parameter**: keine
  - **Erklärung**: Leitet das START-Event (EI) nur dann als EO weiter, wenn PERMIT (aus StartPermit.Q) TRUE ist.
- **MainSR**: Haupt-Speicherglied für den Ausgang
  - **Typ**: adapter::events::unidirectional::AX_SR
  - **Parameter**: keine
  - **Erklärung**: S setzt den Ausgang (nur über StartGate), R setzt ihn zurück (durch STOP oder StopFeedback). Der Adapterausgang Q führt zum Digitalausgang.
- **DigitalOutput_Q1**: logiBUS Digitalausgang
  - **Typ**: logiBUS::io::DQ::logiBUS_QXA
  - **Parameter**: QI = TRUE, Output = Output_Q1
  - **Erklärung**: Gibt den Zustand von MainSR.Q an den physischen Ausgang Q1 weiter.

Programmablauf und Verbindungen

1. START.IND → StartGate.EI und StartGate.EO → MainSR.S (EventConnections): Ein Klick auf I1 setzt den Ausgang nur dann, wenn StartGate freigegeben ist.
2. STOP.IND → MainSR.R (EventConnection): Ein Klick auf I2 setzt den Ausgang sofort zurück.
3. StopFeedback.IND → MainSR.R und StopFeedback.IND → StartPermit.R (EventConnections): Die Hardware-Not-Halt-Rückmeldung an I7 schaltet den Ausgang ab und sperrt zusätzlich START, indem sie den Freigabe-Latch zurücksetzt.
4. ResetInput.IND → ResetButton.REQ (EventConnection) und ResetInput.IN → ResetButton.ResetIn (DataConnection): Pegel und Event von I8 gelangen an SF_ResetButton.
5. ResetButton.minTimeOut → minTimeOut.TimeOutSocket und ResetButton.maxTimeOut → maxTimeOut.TimeOutSocket (AdapterConnections): Die beiden Timer überwachen das gültige Zeitfenster der Tasten-Flanke.
6. ResetButton.EResetOut → StartPermit.S (EventConnection) und StartPermit.Q → StartGate.PERMIT (DataConnection): Eine gültige RESET-Quittierung gibt START wieder frei.
7. MainSR.Q → DigitalOutput_Q1.OUT (AdapterConnection): Der Zustand des Speicherglieds liegt am physischen Ausgang Q1.

Verhalten: START (I1) schaltet Q1 nur ein, wenn eine Freigabe vorliegt. STOP (I2) oder die Not-Halt-Rückmeldung (I7) schalten Q1 aus; die Rückmeldung sperrt zusätzlich START. Nach der Sperre bleibt Q1 auch nach einer gültigen RESET-Quittierung (I8) zunächst aus; erst ein erneuter START-Klick schaltet Q1 wieder ein.

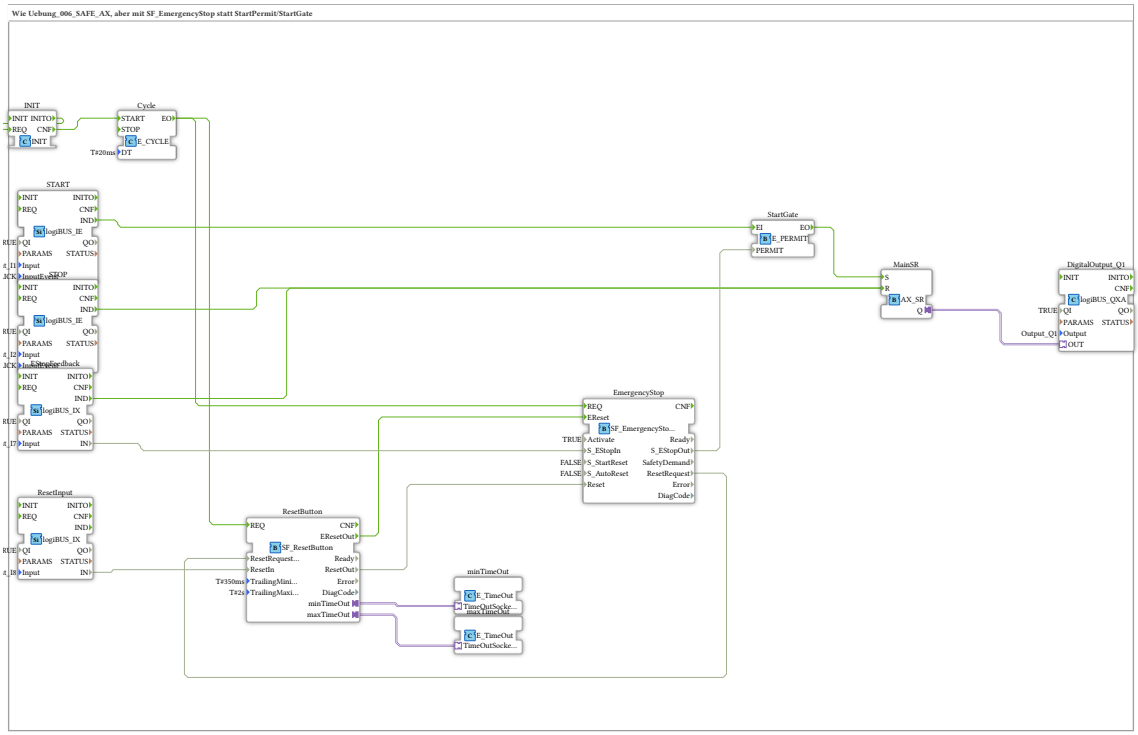
Zusammenfassung

Die Übung erweitert das einfache SR-Flip-Flop aus Uebung_006_AX um eine Wiederanlaufsperre: Der Freigabe-Latch StartPermit und das Gatter StartGate blockieren START nach einer Not-Halt-Rückmeldung, bis SF_ResetButton eine gültige RESET-Flanke erkannt hat. Im Kaltstart ist START ebenfalls gesperrt, weil StartPermit.Q den Typ-Default FALSE hat.

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

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

2.1.2.84 Uebung_006_SAFE_AX_02: Start/Stopp mit SF_EmergencyStop statt StartPermit/StartGate



• — ○ • — ○ • — ○ •

Einleitung

Diese Übung ist die zweite SAFE-Variante von `Uebung_006_AX`. Statt der handgebauten Lösung aus `StartPermit` (E_RS) und `StartGate` (E_PERMIT) in `Uebung_006_SAFE_AX` kommt hier der PLCopen-Standardbaustein `SF_EmergencyStop` (Part 1, Kap. 6.5) zum Einsatz. `SF_EmergencyStop` und `SF_ResetButton` sind im eingebetteten Muster (Figure 13 der Spezifikation) über `ResetRequest`/`ResetRequested` verkoppelt: `SF_EmergencyStop` fordert `SF_ResetButton` nur dann zur Überwachung an, wenn tatsächlich ein Reset nötig ist. Mit `S_StartReset = FALSE` und `S_AutoReset = FALSE` gibt es keinen automatischen Wiederanlauf; der RESET-Taster (`I8`) ist immer zwingend erforderlich.

Verwendete Funktionsbausteine (FBs)

- **START**: Taster START
  - **Typ**: `logiBUS::io::DI::logiBUS_IE`
  - **Parameter**: `QI = TRUE`, `Input = Input_I1`, `InputEvent = BUTTON_SINGLE_CLICK`
  - **Erklärung**: Meldet einen Einzelklick auf `I1` als Event `IND`.
- **STOP**: Hand-Stopp (nicht HW-Not-Halt)
  - **Typ**: `logiBUS::io::DI::logiBUS_IE`
  - **Parameter**: `QI = TRUE`, `Input = Input_I2`, `InputEvent = BUTTON_SINGLE_CLICK`
  - **Erklärung**: Meldet einen Einzelklick auf `I2` als Event `IND`.
- **EStopFeedback**: Rückmeldung des Not-Halt-Kreises (`I7 = TRUE`: Not-Halt-Kreis ok, `I7 = FALSE`: Stopp erkannt)
  - **Typ**: `logiBUS::io::DI::logiBUS_IX`
  - **Parameter**: `QI = TRUE`, `Input = Input_I7`
  - **Erklärung**: Liefert den Pegel `IN` und bei Änderung das Event `IND`.
- **ResetInput**: Pegel und Event für `SF_ResetButton`
  - **Typ**: `logiBUS::io::DI::logiBUS_IX`
  - **Parameter**: `QI = TRUE`, `Input = Input_I8`

  - **Erklärung**: Liefert den Pegel des RESET-Tasters `I8`.
- **ResetButton**: PLCopen-Reset-Taster-Überwachung
  - **Typ**: `logiBUS::safety::PLCopen::SF_ResetButton`
  - **Parameter**: `TrailingMinimum = T#350ms`, `TrailingMaximum = T#2s`
  - **Erklärung**: Wertet den Reset-Taster mit den Zeitgrenzen `TrailingMinimum` und `TrailingMaximum` aus; die Zeitüberwachung läuft über die Adapter `minTimeOut`/`maxTimeOut`.
- **EmergencyStop**: PLCopen-Not-Halt
  - **Typ**: `logiBUS::safety::PLCopen::SF_EmergencyStop`
  - **Parameter**: `Activate = TRUE`, `S_StartReset = FALSE`, `S_AutoReset = FALSE`
  - **Erklärung**: Wertet die Not-Halt-Rückmeldung aus und liefert die Freigabe `S_EStopOut`. Ohne Startreset und ohne Autoreset ist immer ein Reset über `I8` nötig.
- **StartGate**: Lässt START nur bei Freigabe durch
  - **Typ**: `iec61499::events::E_PERMIT`
  - **Parameter**: keine
  - **Erklärung**: Reicht das Event vom `START`-Taster nur weiter, wenn `PERMIT` (aus `EmergencyStop.S_EStopOut`) gesetzt ist.
- **MainSR**: Haupt-Set/Reset-Speicher
  - **Typ**: `adapter::events::unidirectional::AX_SR`
  - **Parameter**: keine
  - **Erklärung**: Wird über `S` gesetzt und über `R` zurückgesetzt; der Zustand `Q` geht als Adapter an den Ausgang.
- **DigitalOutput_Q1**: logiBUS Digitalausgang
  - **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
  - **Parameter**: `QI = TRUE`, `Output = Output_Q1`
  - **Erklärung**: Legt den Zustand von `MainSR.Q` auf den physischen Ausgang `Q1`.
- **INIT**: Startimpuls
  - **Typ**: `iec61131::booleanOperators::INIT`
  - **Parameter**: keine
  - **Erklärung**: `INIT0` ist auf `REQ` zurückgeführt, `CNF` startet den zyklischen Baustein `Cycle`.
- **Cycle**: Zyklische Auswertung von `SF_EmergencyStop` und `SF_ResetButton`
  - **Typ**: `iec61499::events::E_CYCLE`
  - **Parameter**: `DT = T#20ms`
  - **Erklärung**: Erzeugt alle 20 ms ein Event `EO` für `EmergencyStop.REQ` und `ResetButton.REQ`.
- **minTimeOut**: Zeitgeber für `TrailingMinimum`
  - **Typ**: `iec61499::events::E_TimeOut`
  - **Parameter**: keine
  - **Erklärung**: Über den Adapter `TimeOutSocket` mit `ResetButton.minTimeOut` verbunden.
- **maxTimeOut**: Zeitgeber für `TrailingMaximum`
  - **Typ**: `iec61499::events::E_TimeOut`
  - **Parameter**: keine
  - **Erklärung**: Über den Adapter `TimeOutSocket` mit `ResetButton.maxTimeOut` verbunden.

Programmablauf und Verbindungen

1. `START.IND` → `StartGate.EI` und `StartGate.EO` → `MainSR.S` (EventConnections): Ein START-Klick setzt den Speicher nur, wenn `StartGate` freigegeben ist.
2. `EmergencyStop.S_EStopOut` → `StartGate.PERMIT` (DataConnection): Die Freigabe des Not-Halt-Bausteins entscheidet über `StartGate`.
3. `STOP.IND` → `MainSR.R` und `EStopFeedback.IND` → `MainSR.R` (EventConnections): Sowohl der Hand-Stopp (`I2`) als auch jede Änderung der Not-Halt-Rückmeldung (`I7`) setzen den Speicher zurück.
4. `EStopFeedback.IN` → `EmergencyStop.S_EStopIn` (DataConnection): Der Pegel des Not-Halt-Kreises wird von `SF_EmergencyStop` ausgewertet.
5. `INIT.INIT0` → `INIT.REQ`, `INIT.CNF` → `Cycle.START`, `Cycle.EO` → `EmergencyStop.REQ` und `Cycle.EO` → `ResetButton.REQ` (EventConnections, im Netzwerkbild ausgeblendet): Nach dem Start werden beide Sicherheitsbausteine zyklisch alle 20 ms angestoßen.
6. `ResetInput.IN` → `ResetButton.ResetIn` und `ResetButton.ResetOut` → `EmergencyStop.Reset` (DataConnections): Der RESET-Taster `I8` läuft über `SF_ResetButton` in den Not-Halt-Baustein.
7. `ResetButton.EResetOut` → `EmergencyStop.EReset` (EventConnection): Das Reset-Event wird an `SF_EmergencyStop` weitergegeben.

8. `EmergencyStop.ResetRequest` ⟶ `ResetButton.ResetRequested` (DataConnection): Die Verkopplung nach Figure 13 – `SF_ResetButton` überwacht nur dann, wenn ein Reset angefordert ist.
9. `ResetButton.minTimeOut` ⟶ `minTimeOut.TimeOutSocket` und `ResetButton.maxTimeOut` ⟶ `maxTimeOut.TimeOutSocket` (AdapterConnections): Die Zeitüberwachung des Reset-Tasters.
10. `MainSR.Q` ⟶ `DigitalOutput_Q1.OUT` (AdapterConnection): Der Speicherzustand steuert den Ausgang `Q1`.

Zusammenfassung

Die Übung 006_SAFE_AX_02 ersetzt die handgebaute Start-Freigabe aus `StartPermit` und `StartGate` durch den PLCopen-Standardbaustein `SF_EmergencyStop` samt `SF_ResetButton`. Weil weder Startreset noch Autoreset aktiv sind, ist nach einem erkannten Stopp immer ein bewusster RESET über `I8` erforderlich, bevor `START` den Ausgang `Q1` wieder setzen kann.

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

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

2.1.2.85 Uebung_006a2_AX: 2x SR und T-Flip-Flop mit IE



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_006a2_AX`.

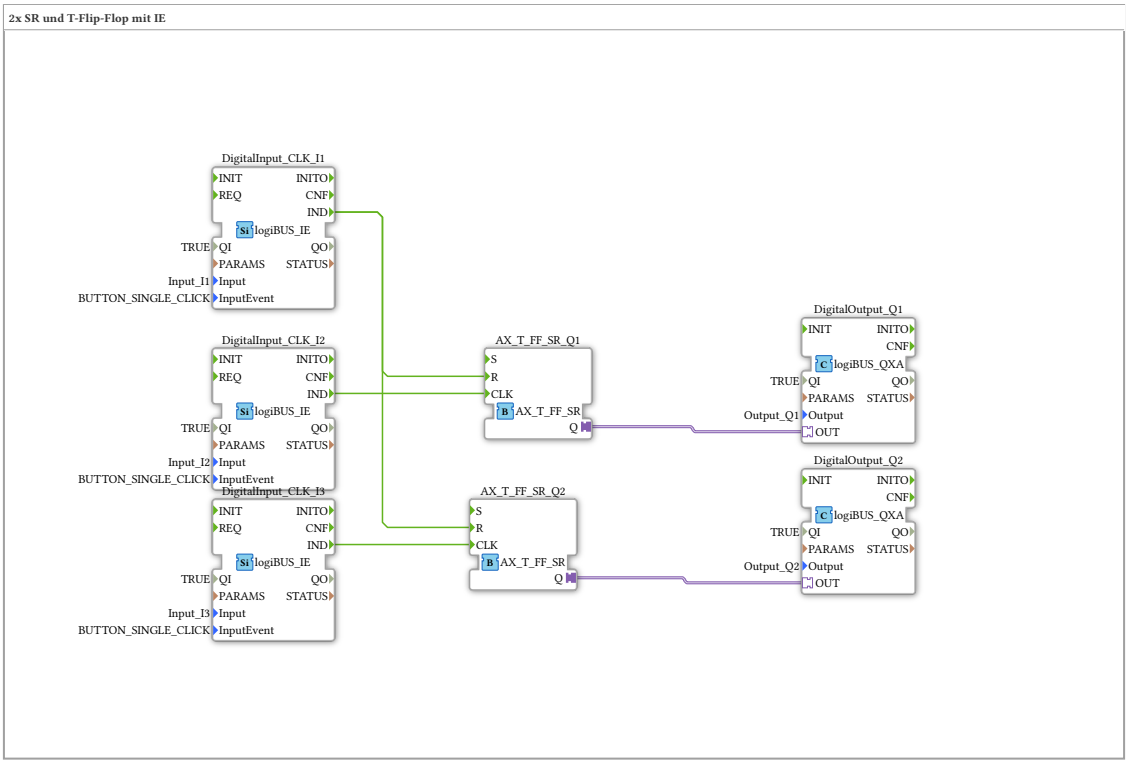
Ziel der Übung

Demonstration einer Zentral-Aus-Funktion.

Beschreibung und Komponenten

Die Subapplikation `Uebung_006a2_AX.SUB` steuert zwei unabhängige Lampen, die gemeinsam gelöscht werden können.

Funktionsbausteine (FBs)



- **`I1`**: Reset für beide.

- **I2**: Toggelt Lampe 1.
- **I3**: Toggelt Lampe 2.
- **2x AX_T_FF_SR**: Je einer pro Lampe.

Funktionsweise

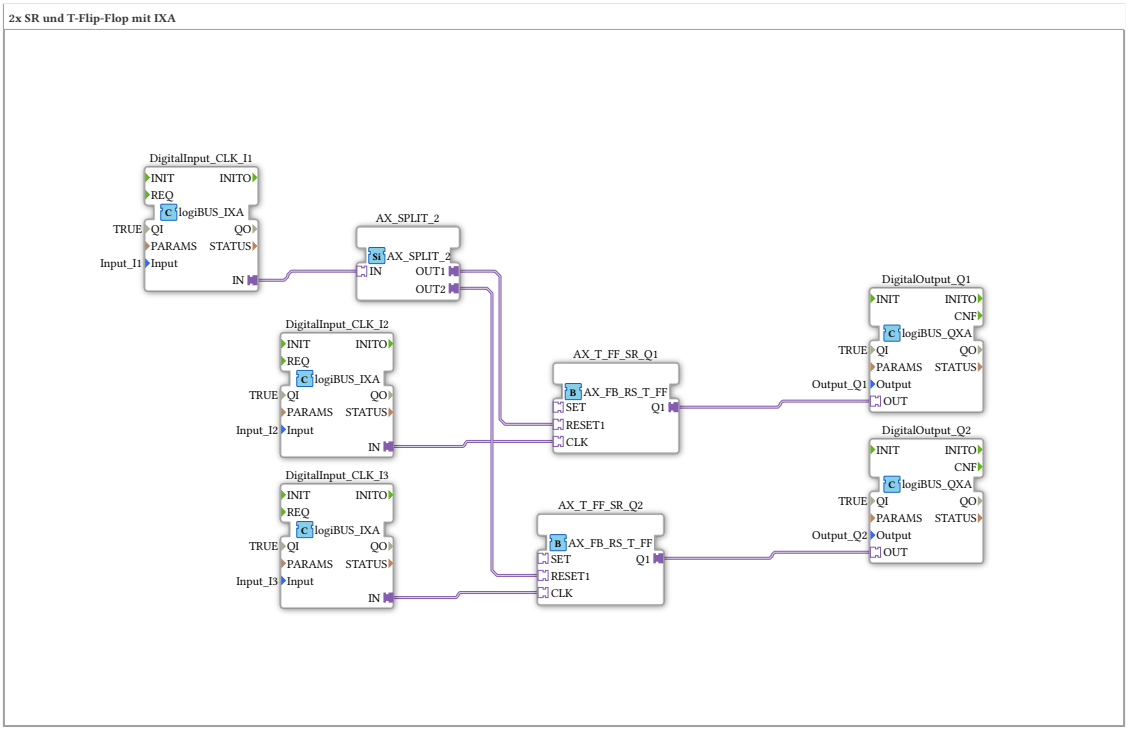
- I2 ist mit CLK von FF1 verbunden.
- I3 ist mit CLK von FF2 verbunden.
- I1 ist mit R von **beiden** Flip-Flops verbunden (Fan-Out).

Ein Druck auf I1 schaltet sofort beide Lampen aus, egal in welchem Zustand sie waren.

Anwendungsbeispiel

Bürobeleuchtung: Jeder Schreibtisch hat sein eigenes Licht (I2, I3), aber am Ausgang gibt es einen Schalter „Raum verlassen“, der alles ausschaltet (I1).

2.1.2.86 Uebung_006a2b_AX: 2x SR und T-Flip-Flop mit IXA



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung zeigt die Anwendung von SR- und T-Flipflops in Kombination mit einem gemeinsamen Rücksetzsignal („Hausmeister-Aus“). Zwei Digitaleingänge (I2, I3) schalten die beiden Ausgänge (Q1, Q2) jeweils um (Toggle). Ein dritter Eingang (I1) setzt beide Ausgänge gleichzeitig zurück. Dadurch wird das Zusammenspiel bistabiler Elemente und die Ereignisverteilung in 4diac veranschaulicht.

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine

DigitalInput_CLK_I1

- **Typ**: `logiBUS::io::DI::logiBUS_IXA`
- **Parameter**: `Input = Input_I1, QI = TRUE`
- **Funktionsweise**: Stellt den digitalen Tastereingang I1 (Resettaster) als Ereignis zur Verfügung. Jede Betätigung erzeugt ein Ereignis am Ausgang `IN`.

DigitalInput_CLK_I2

- **Typ**: `logiBUS::io::DI::logiBUS_IXA`
- **Parameter**: `Input = Input_I2`, `QI = TRUE`
- **Funktionsweise**: Stellt den digitalen Tastereingang I2 (Toggle von Q1) als Ereignis zur Verfügung.

DigitalInput_CLK_I3

- **Typ**: `logiBUS::io::DI::logiBUS_IXA`
- **Parameter**: `Input = Input_I3`, `QI = TRUE`
- **Funktionsweise**: Stellt den digitalen Tastereingang I3 (Toggle von Q2) als Ereignis zur Verfügung.

AX_SPLIT_2

- **Typ**: `adapter::events::unidirectional::AX_SPLIT_2`
- **Funktionsweise**: Verteilt ein eingehendes Ereignis (von I1) auf zwei Ausgänge (`OUT1`, `OUT2`). So kann ein einziges Tastersignal gleichzeitig an mehrere Empfänger gesendet werden.

AX_T_FF_SR_Q1

- **Typ**: `adapter::bistableElements::AX_FB_RS_T_FF`
- **Funktionsweise**: Kombinierter RS- und T-Flipflop.
  - Am Ereigniseingang `CLK` (verbunden mit I2) wird der Ausgang `Q1` bei jeder positiven Flanke umgeschaltet (Toggle).
  - Der Ereigniseingang `RESET1` (verbunden mit `AX_SPLIT_2.OUT1`) setzt den Ausgang `Q1` zurück.

AX_T_FF_SR_Q2

- **Typ**: `adapter::bistableElements::AX_FB_RS_T_FF`
- **Funktionsweise**: Gleichartiger Flipflop wie oben.
  - `CLK` von I3, `RESET1` von `AX_SPLIT_2.OUT2`.
  - Steuert den Ausgang `Q2`.

DigitalOutput_Q1

- **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
- **Parameter**: `Output = Output_Q1`, `QI = TRUE`
- **Funktionsweise**: Digitalausgang, der den Zustand von `Q1` auf den physikalischen Ausgang `Output_Q1` ausgibt.

DigitalOutput_Q2

- **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
- **Parameter**: `Output = Output_Q2`, `QI = TRUE`
- **Funktionsweise**: Digitalausgang, der den Zustand von `Q2` auf den physikalischen Ausgang `Output_Q2` ausgibt.

Programmablauf und Verbindungen

Die Schaltung arbeitet nach folgendem Prinzip:

1. **T-Flipflop-Betrieb**: Jede Betätigung von Taster I2 erzeugt ein Ereignis am `CLK`-Eingang von `AX_T_FF_SR_Q1`. Der Flipflop wechselt seinen Ausgangszustand (Toggle). Entsprechendes gilt für Taster I3 und `AX_T_FF_SR_Q2`.
2. **Zentraler Reset („Hausmeister-Aus“)**: Wird Taster I1 gedrückt, wird das Ereignis über den Splitter `AX_SPLIT_2` auf die `RESET1`-Eingänge beider Flipflops verteilt. Beide Ausgänge (`Q1`, `Q2`) werden sofort zurückgesetzt.
3. **Ausgabe**: Die internen Zustände von `Q1` und `Q2` werden über die Digitalausgänge `Output_Q1` und `Output_Q2` nach außen geführt.

Lernziele:

- Verständnis des Verhaltens von T-Flipflops und RS-Flipflops.
- Ereignisbasierte Programmierung und Weiterleitung mit `AX_SPLIT_2`.
- Einfaches Zusammenspiel von digitalen Ein- und Ausgängen.

Voraussetzungen:

- Grundlegende Kenntnisse der 4diac-IDE und der Ereignissteuerung nach IEC 61499.

Ausführung: Die Übung kann direkt in einer 4diac-Laufzeitumgebung (z. B. FORTE) mit entsprechend konfigurierten logiBUS-I/O-Modulen gestartet werden. Taster I2 und I3 schalten die Ausgänge, Taster I1 setzt alles zurück.

Zusammenfassung

Die Übung „Uebung_006a2b_AX“ implementiert zwei unabhängige T-Flipflops, die über einen gemeinsamen Rücksetzeingang verfügen. Durch die Verwendung der Adapter-FBs `AX_FB_RS_T_FF` und `AX_SPLIT_2` wird eine kompakte und leicht nachvollziehbare Steuerung realisiert. Der Schwerpunkt liegt auf dem Verständnis bistabiler Schaltungen und der ereignisgesteuerten Kommunikation in 4diac.

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

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

2.1.2.87 Uebung_006a3_AX: SR und T-Flip-Flop mit 3x IE; links/rechtslauf



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_006a3_AX`. Dies ist eine komplexere Anwendung zur Ansteuerung eines Motors mit zwei Drehrichtungen.

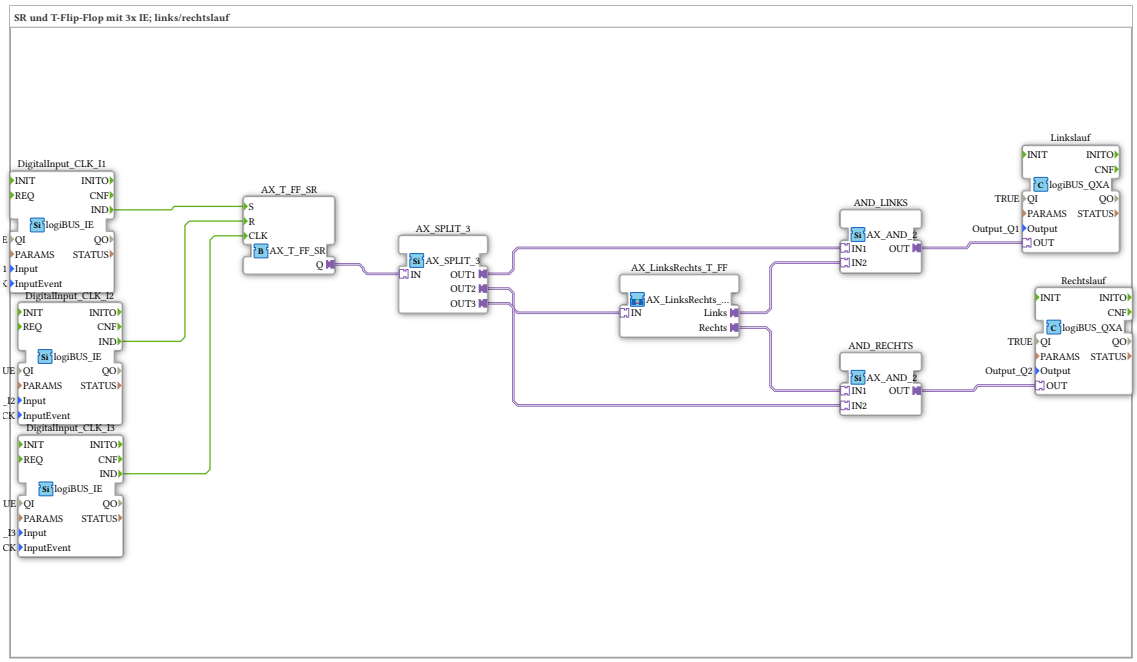
Ziel der Übung

Realisierung einer Wende-Schütz-Steuerung mit Software-Verriegelung. Es darf niemals gleichzeitig „Links“ und „Rechts“ angesteuert werden, da dies einen Kurzschluss im Leistungskreis verursachen würde.

Beschreibung und Komponenten

Die Subapplikation `Uebung_006a3_AX.SUB` nutzt eine Kombination aus Flip-Flop, Splitter und einer custom Sub-App (`Uebung_006a3_sub_AX`).

Funktionsbausteine (FBs)



- **`I1` (Set)**: Einschalten (in die zuletzt gewählte Richtung oder Default).
- **`I2` (Reset)**: Ausschalten.
- **`I3` (Toggle)**: Start/Stopp.
- **`AX_T_FF_SR`**: Der Hauptspeicher „Motor Ein/Aus“.

- **AX_SPLIT_3**: Verteilt das „Motor ist an“-Signal.
- **AX_LinksRechts_T_FF (SubApp)**: Speichert die aktuelle *Richtung* (Links oder Rechts).
- **2x AX_AND_2**: Verriegelungsgatter.

Funktionsweise

1. Das AX_T_FF_SR bestimmt, ob der Motor überhaupt laufen soll.
2. Die SubApp AX_LinksRechts_T_FF ist ein Richtungs-Speicher (Toggle). Jedes Mal, wenn der Motor eingeschaltet wird (Event von SPLIT_3.OUT1), toggelt diese SubApp die Richtung für den *nächsten* oder *aktuellen* Lauf (abhängig von der genauen internen Verschaltung).
3. Die UND-Gatter verknüpfen „Motor Ein“ (SPLIT_3) mit „Richtung Links“ bzw. „Richtung Rechts“.
4. Dadurch ist sichergestellt, dass immer nur ein Ausgang (Q1 oder Q2) aktiv ist.

Hinweis: Die genaue Logik des Richtungwechsels (bei jedem Start? oder durch extra Taste?) hängt von der Uebung_006a3_sub_AX ab.

Anwendungsbeispiel

Waschmaschine oder **Rührwerk**: Der Motor soll abwechselnd links und rechts laufen, aber natürlich nie gleichzeitig.

2.1.2.88 Uebung_006a3_sub_AX

Hinweis

Diese Übung existiert in den aktuellen Trainingsquellen nicht mehr (entfernt, zusammengelegt oder umbenannt). Diese Seite bleibt nur als Platzhalter für bestehende Verweise erhalten. Die aktuellen Übungen stehen in der Übungsübersicht.

2.1.2.89 Uebung_006a4_AX: SR und T-Flip-Flop mit 3x IE



Dieser Artikel beschreibt die logiBUS®-Übung Uebung_006a4_AX. Sie ist eine Optimierung von Uebung_006a3_AX durch Verwendung eines vorgefertigten Bausteins.

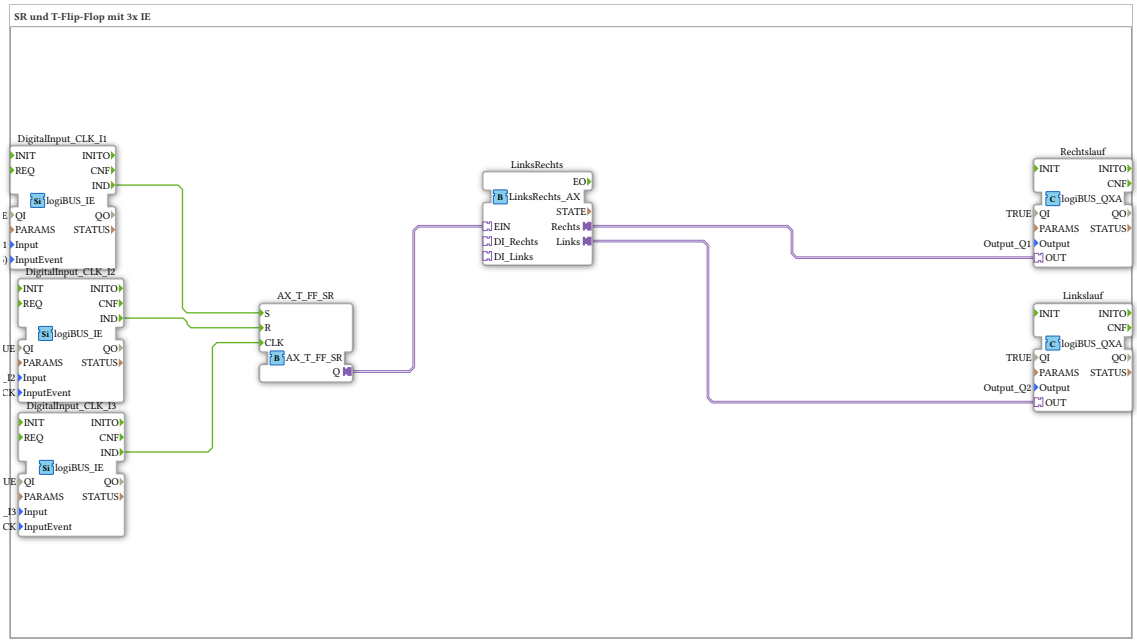
Ziel der Übung

Nutzung von Bibliotheken („Don’t reinvent the wheel“).

Beschreibung und Komponenten

Die Subapplikation Uebung_006a4_AX.SUB ersetzt das komplexe Netzwerk aus Gatter und SubApp der vorherigen Übung durch den Baustein LinksRechts_AX.

Funktionsbausteine (FBs)



- **`LinksRechts`**: Typ `logiBUS::utils::sequence::verteiler::LinksRechts_AX`. Dieser Baustein kapselt die komplette Logik für die Richtungssteuerung und Verriegelung.
- **`AX_T_FF_SR`**: Liefert weiterhin das „Ein/Aus“ Signal an den Eingang `EIN` des Verteilers.

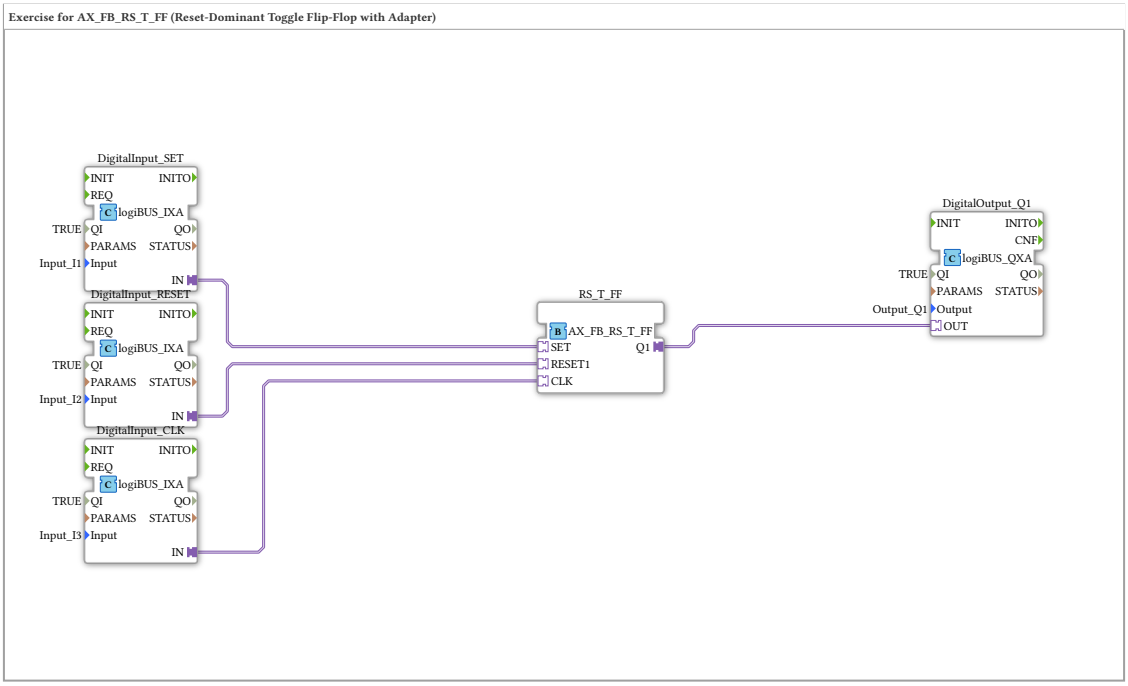
Funktionsweise

Die Logik ist gekapselt. Der Baustein `LinksRechts_AX` kümmert sich intern darum, das Eingangssignal abwechselnd auf den Ausgang `Links` und `Rechts` zu leiten.

Vorteil

Der Code ist wesentlich aufgeräumter, lesbarer und weniger fehleranfällig.

2.1.2.90 Uebung_006a4_AX_RS: Exercise for AX_FB_RS_T_FF (Reset-Dominant Toggle Flip-Flop with Adapter)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert den Einsatz des **Reset-dominanten Toggle-Flipflops (AX_FB_RS_T_FF)** mit Adapter-Schnittstelle in der 4diac-IDE. Das Flipflop besitzt drei Eingänge (SET, RESET1, CLK) und einen Ausgang (Q1). Die Schaltung wird über digitale logiBUS-Eingänge (Input_I1 als SET, Input_I2 als RESET, Input_I3 als Takt) gesteuert. Das Ausgangssignal wird auf den logiBUS-Ausgang Output_Q1 gelegt. Ziel ist es, das Verhalten eines **reset-dominanten** Toggle-Flipflops zu verstehen und die Verdrahtung mit Adapter-FBs nachzuvollziehen.

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_SET** (Typ: `logiBUS_IXA`): Liest den logiBUS-Eingang `Input_I1` (SET-Signal).
- **DigitalInput_RESET** (Typ: `logiBUS_IXA`): Liest den logiBUS-Eingang `Input_I2` (RESET-Signal).
- **DigitalInput_CLK** (Typ: `logiBUS_IXA`): Liest den logiBUS-Eingang `Input_I3` (Taktsignal).
- **RS_T_FF** (Typ: `AX_FB_RS_T_FF`): Reset-dominantes Toggle-Flipflop mit Adapter-Schnittstelle.
- **DigitalOutput_Q1** (Typ: `logiBUS_QXA`): Gibt den Zustand des Flipflops an den logiBUS-Ausgang `Output_Q1` aus.

Parameter

FB	Parameter	Wert
DigitalInput_SET	QI	TRUE
DigitalInput_SET	Input	Input_I1
DigitalInput_RESET	QI	TRUE
DigitalInput_RESET	Input	Input_I2
DigitalInput_CLK	QI	TRUE
DigitalInput_CLK	Input	Input_I3
DigitalOutput_Q1	QI	TRUE

DigitalOutput_Q1	`Output`	`Output_Q1`

Programmablauf und Verbindungen

Die logiBUS-Eingänge werden über die Funktionsbausteine `DigitalInput_SET`, `DigitalInput_RESET` und `DigitalInput_CLK` ausgelesen und als Adapter-Sockets an das Flipflop `RS_T_FF` weitergegeben.

Verbindungen (AdapterConnections):

- `DigitalInput_SET.IN` ⟶ `RS_T_FF.SET`
- `DigitalInput_RESET.IN` ⟶ `RS_T_FF.RESET1`
- `DigitalInput_CLK.IN` ⟶ `RS_T_FF.CLK`
- `RS_T_FF.Q1` ⟶ `DigitalOutput_Q1.OUT`

Funktionsweise des Flipflops:

- Bei einer steigenden Flanke am CLK-Eingang toggelt der Ausgang Q1 (d. h. er wechselt seinen Zustand von FALSE zu TRUE oder umgekehrt).
- Ist der RESET1-Eingang aktiv (TRUE), wird der Ausgang **sofort und dominant** auf FALSE gesetzt – unabhängig vom aktuellen Zustand und vom Taktsignal.
- Der SET-Eingang setzt den Ausgang auf TRUE, sofern kein RESET anliegt und kein Taktimpuls ausgeführt wird. Da RESET dominant ist, hat RESET stets Vorrang.

Lernziele:

- Verständnis der Funktionsweise eines reset-dominanten Toggle-Flipflops.
- Umgang mit Adapter-basierten Funktionsbausteinen in 4diac.
- Einbindung logiBUS-Hardware-Eingänge/-Ausgänge in ein Automatisierungsprojekt.

Schwierigkeitsgrad: Mittel **Vorkenntnisse:** Grundlegende Kenntnisse über Flipflops und die 4diac-IDE.

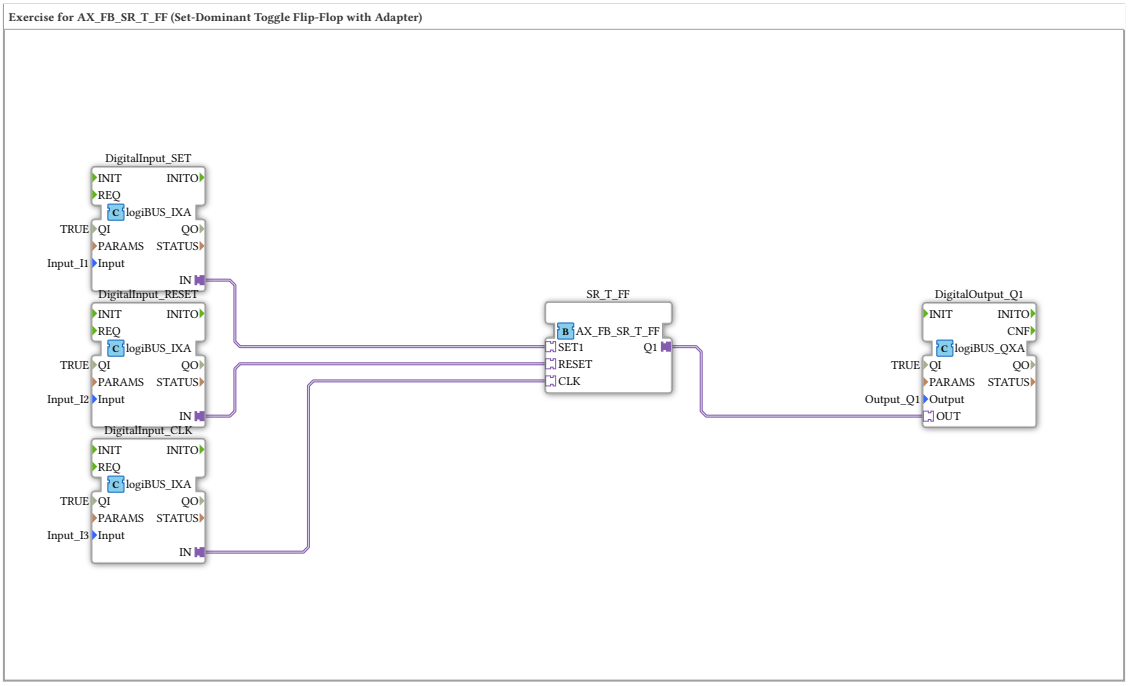
Zusammenfassung

Die Übung `Uebung_006a4_AX_RS` realisiert ein reset-dominantes Toggle-Flipflop mit drei logiBUS-Eingängen und einem Ausgang. Durch die Adapter-Schnittstelle des Flipflops wird eine klare, funktionale Verbindung zwischen Hardware-Ein-/Ausgängen und der Logik des Bausteins erreicht. Die dominante RESET-Funktion sorgt für ein sicheres Grundverhalten in Steuerungsanwendungen.

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

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

2.1.2.91 Uebung_006a4_AX_SR: Exercise for AX_FB_SR_T_FF (Set-Dominant Toggle Flip-Flop with Adapter)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung führt in den Funktionsbaustein `AX_FB_SR_T_FF` (Set-Dominant Toggle Flip-Flop) ein, der über Adapter angebunden wird. Ziel ist es, das Verhalten eines setzdominanten Toggle-Flipflops zu verstehen und in einer einfachen Steuerung zu testen.

Der Baustein wird mit drei digitalen Eingängen (SET, RESET, CLK) und einem digitalen Ausgang (Q1) verschaltet. Durch die Verbindung mit den logiBUS-Ein-/Ausgangsbausteinen kann die Schaltung direkt an einer realen Hardware getestet werden.

Verwendete Funktionsbausteine (FBs)

In dieser Übung werden folgende Funktionsbausteine eingesetzt:

FB-Name	Typ	Parameter
`DigitalInput_SET`	`logiBUS::io::DI::logiBUS_IXA`	Input = `Input_I1`
`DigitalInput_RESET`	`logiBUS::io::DI::logiBUS_IXA`	Input = `Input_I2`
`DigitalInput_CLK`	`logiBUS::io::DI::logiBUS_IXA`	Input = `Input_I3`
`SR_T_FF`	`adapter::bistableElements::AX_FB_SR_T_FF`	(keine Parameter)
`DigitalOutput_Q1`	`logiBUS::io::DQ::logiBUS_QXA`	Output = `Output_Q1`

- **`logiBUS_IXA`**: Digitaler Eingang, der das Signal von einem logiBUS-Eingangskanal über den Adapter einspeist.
- **`AX_FB_SR_T_FF`**: Set-dominantes Toggle-Flipflop mit den Adapter-Schnittstellen `SET1`, `RESET` und `CLK`. Der Ausgang `Q1` toggelt bei jeder positiven Flanke am Clock-Eingang, wenn **SET** aktiv ist; bei aktivem **RESET** wird der Ausgang zurückgesetzt.
- **`logiBUS_QXA`**: Digitaler Ausgang, der das Signal an einen logiBUS-Ausgangskanal weiterleitet.

Es werden keine weiteren Sub-Applikationen (`SubAppType`) innerhalb des Netzwerks verwendet.

Programmablauf und Verbindungen

Der logische Ablauf der Übung ist wie folgt:

1. **Eingänge**: Die physikalischen Eingänge `Input_I1` (SET), `Input_I2` (RESET) und `Input_I3` (CLK) werden über die drei `logiBUS_IXA`-Bausteine in die 4diac-Umgebung eingelesen.
2. **Flip-Flop**: Die Signale gelangen über Adapterverbindungen zu `SR_T_FF`:
   - `DigitalInput_SET.IN` ⟶ `SR_T_FF.SET1`
   - `DigitalInput_RESET.IN` ⟶ `SR_T_FF.RESET`
   - `DigitalInput_CLK.IN` ⟶ `SR_T_FF.CLK`
3. **Ausgang**: Der Ausgang `SR_T_FF.Q1` wird auf den Digitalausgang `DigitalOutput_Q1.OUT` übertragen und an `Output_Q1` ausgegeben.

Funktionsweise des SR_T_FF:

- Bei einem aktiven **Reset** (`RESET = 1`) wird der Ausgang sofort auf `FALSE` gesetzt.
- Ist **Reset** inaktiv und **SET** aktiv, toggelt der Ausgang bei jeder positiven Flanke an `CLK`. (Set-dominant bedeutet, dass ein gleichzeitig aktiver Set die Toggle-Funktion zulässt; bei inaktivem Set wird der Ausgang nicht getoggelt.)
- Sind beide, Set und Reset, inaktiv, bleibt der Ausgang unverändert.

Lernziele:

- Verständnis des setzdominanten Toggle-Flipflops und seiner Adapter-Schnittstelle.
- Einbindung von Hardware-Ein-/Ausgängen über logiBUS-Adapter.
- Analyse des zeitlichen Verhaltens bei unterschiedlichen Eingangskombinationen.

Schwierigkeitsgrad: Einfach **Vorkenntnisse**: Grundlegender Umgang mit der 4diac-IDE, einfache digitale Logik.

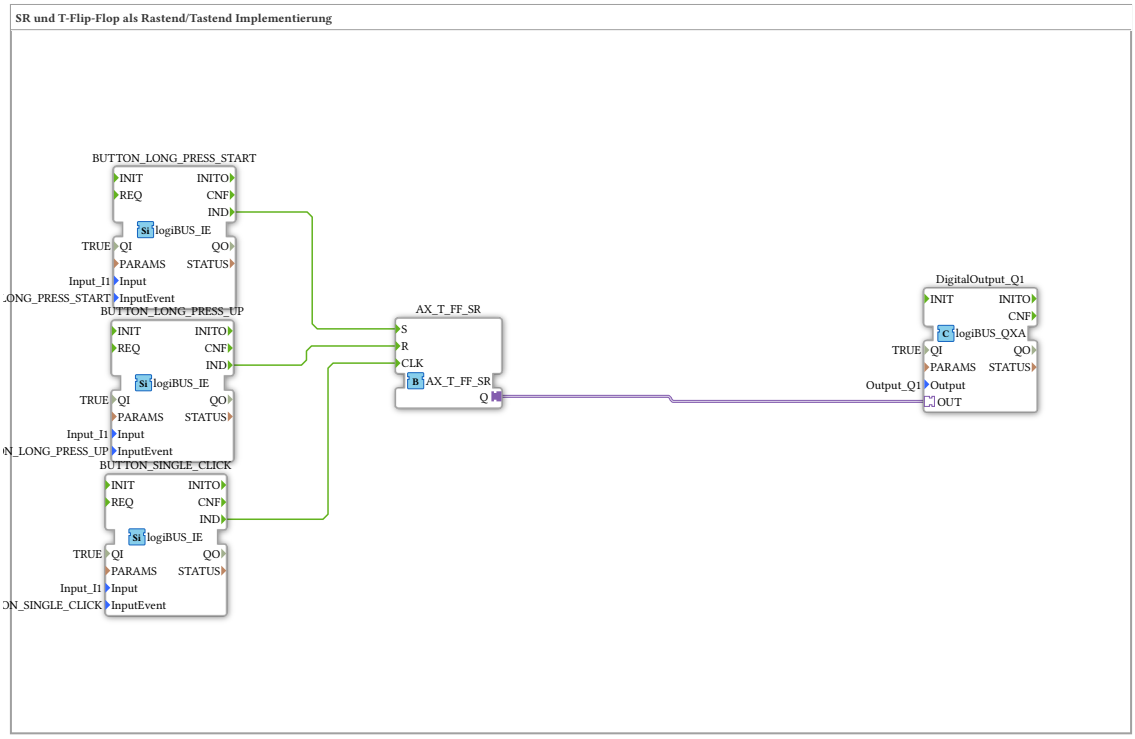
Zusammenfassung

Die Übung `Uebung_006a4_AX_SR` demonstriert die Verwendung des setzdominanten Toggle-Flipflops `AX_FB_SR_T_FF` in einer 4diac-Umgebung. Durch die klare Trennung von Ein-/Ausgangsadaptern und dem eigentlichen Flipflop-Baustein wird eine hardwarenahe Steuerung realisiert, die sich direkt auf einer logiBUS-Plattform testen lässt. Der Schwerpunkt liegt auf dem Verständnis der Toggle-Funktion unter Berücksichtigung des dominanten Set- und Reset-Verhaltens.

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

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

2.1.2.92 Uebung_006a5_AX: SR und T-Flip-Flop als Rastend/Tastend-Implementierung



• – ○ • – ○ • – ○ •

Einleitung

Die Übung `Uebung_006a5_AX` zeigt, wie sich mit **einem einzigen physischen Taster** (`Input_I1`) sowohl ein **rastendes** (schaltendes) als auch ein **tastendes** (umschaltendes) Bedienverhalten realisieren lässt. Anders als `Uebung_006a_AX`, die für Set, Reset und Toggle drei getrennte Taster (`I1`, `I2`, `I3`) benötigt, wertet diese Übung an einem einzigen Eingang unterschiedliche Tasten-Gesten aus – langes Drücken, Loslassen nach langem Drücken und kurzer Klick – und speist diese in den Baustein `AX_T_FF_SR` ein.

Damit lässt sich mit einem einzigen Taster ein Ausgang:

- **fest einschalten** (Halten/lange drücken → *rastend* wie ein Schalter),
- **fest ausschalten** (Loslassen nach langem Drücken),
- **umschalten** (kurzer Klick → *tastend* wie ein klassischer Taster).

Verwendete Funktionsbausteine (FBs)

FB-Name	Typ	Parameter
`BUTTON_LONG_PRESS_START`	`logiBUS::io::DI::logiBUS_IE`	Input = `Input_I1`, InputEvent = `BUTTON_LONG_PRESS_START`
`BUTTON_LONG_PRESS_UP`	`logiBUS::io::DI::logiBUS_IE`	Input = `Input_I1`, InputEvent = `BUTTON_LONG_PRESS_UP`
`BUTTON_SINGLE_CLICK`	`logiBUS::io::DI::logiBUS_IE`	Input = `Input_I1`, InputEvent = `BUTTON_SINGLE_CLICK`
`AX_T_FF_SR`	`adapter::events::unidirectional::AX_T_FF_SR`	(keine Parameter)
`DigitalOutput_Q1`	`logiBUS::io::DQ::logiBUS_QXA`	Output = `Output_Q1`

- **`logiBUS_IE`**: Event-Eingangsbaustein, der aus dem rohen Tastersignal von `Input_I1` erkannte Tastengesten (z. B. Klick, Doppelklick, langer Druck) als Events ausgibt. Über den Parameter `InputEvent` wird festgelegt, auf welche konkrete Geste der jeweilige FB reagiert. Der Ausgang `IND` feuert, sobald die konfigurierte Geste erkannt wurde. Alle drei FBs in dieser Übung hören auf **denselben** physischen Eingang `Input_I1`, filtern aber jeweils eine andere Geste heraus.

- **AX_T_FF_SR**: Vereint Set (S), Reset (R) und Toggle (CLK) in einem einzigen Baustein mit Adapter-Ausgang Q. S setzt den Ausgang fest auf TRUE, R setzt ihn fest auf FALSE, CLK toggelt den aktuellen Zustand.
- **logiBUS_QXA**: Digitaler Ausgang, der das Adapter-Signal Q an den physischen Ausgang Output_Q1 weiterleitet.

Zusätzlich enthält das Netzwerk einen reinen Kommentar-Block („Universal Eingang: so können wir mit einem Taster ODER einem Schalter arbeiten.“) als Hinweis, dass dieselbe Verschaltung sowohl mit einem tastenden Taster als auch mit einem rastenden Schalter am physischen Eingang funktioniert.

Programmablauf und Verbindungen

1. **Gestenerkennung**: Die drei logiBUS_IE-Bausteine lesen alle denselben Eingang Input_I1, erkennen aber jeweils eine eigene Geste:
   - BUTTON_LONG_PRESS_START → feuert, sobald der Taster lange genug gehalten wurde (Beginn des Langdrucks).
   - BUTTON_LONG_PRESS_UP → feuert, wenn der Taster nach einem langen Druck losgelassen wird.
   - BUTTON_SINGLE_CLICK → feuert bei einem kurzen, einzelnen Klick.
2. **Verschaltung mit dem Flip-Flop** (EventConnections):
   - BUTTON_LONG_PRESS_START.IND → AX_T_FF_SR.S (Langer Druck **beginnt** → Ausgang wird gesetzt)
   - BUTTON_LONG_PRESS_UP.IND → AX_T_FF_SR.R (Loslassen nach langem Druck → Ausgang wird zurückgesetzt)
   - BUTTON_SINGLE_CLICK.IND → AX_T_FF_SR.CLK (Kurzer Klick → Ausgang wird umgeschaltet)
3. **Ausgabe** (AdapterConnections): AX_T_FF_SR.Q → DigitalOutput_Q1.OUT → physischer Ausgang Output_Q1.

Funktionsweise des AX_T_FF_SR

Der Baustein AX_T_FF_SR besitzt intern die Zustände START, SET und RESET:

- Im Zustand SET ist Q = TRUE, im Zustand RESET ist Q = FALSE.
- Ein S-Event schaltet immer in den Zustand SET (Q := TRUE).
- Ein R-Event schaltet immer in den Zustand RESET (Q := FALSE).
- Ein CLK-Event **toggelt** zwischen SET und RESET (aus SET wird RESET und umgekehrt).

Übertragen auf die Bedienung mit **einem** Taster ergibt das:

- **Halten (lange drücken)** → S → Ausgang geht sicher auf TRUE (funktioniert wie ein *rastender* Schalter, der „Ein“ einrastet).
- **Loslassen nach langem Halten** → R → Ausgang geht sicher auf FALSE.
- **Kurz klicken** → CLK → Ausgang wechselt seinen Zustand (funktioniert wie ein klassischer *tastender* Umschalter).

Damit lassen sich mit ein und demselben Taster sowohl ein eindeutiges „Ein“/„Aus“ (über langen Druck) als auch ein bequemes Umschalten (über kurzen Klick) erreichen, ohne dass dafür mehrere Taster nötig sind.

Anwendungsbeispiel

Smart-Home-Lichtsteuerung mit nur einem Taster:

- Kurzer Klick an der Wand: Licht ein-/ausschalten (Toggle über CLK).
- Taster lange gedrückt halten: Licht garantiert einschalten, unabhängig vom vorherigen Zustand (S) – z. B. für Besucher, die den aktuellen Zustand nicht kennen.
- Loslassen nach langem Halten: Licht garantiert ausschalten (R) – z. B. beim Verlassen des Raums.

Lernziele

- Verständnis, wie unterschiedliche Tastengesten (BUTTON_LONG_PRESS_START, BUTTON_LONG_PRESS_UP, BUTTON_SINGLE_CLICK) aus **einem** physischen Eingang mehrere logische Events erzeugen.
- Verständnis des kombinierten Set/Reset/Toggle-Verhaltens von AX_T_FF_SR.
- Erkennen des Vorteils gegenüber Uebung_006a_AX: identisches Set/Reset/Toggle-Verhalten, aber mit nur einem statt drei physischen Tastern.
- Unterscheidung zwischen *rastendem* (haltendem) und *tastendem* (impulsartigem) Bedienverhalten an ein und demselben Taster.

Schwierigkeitsgrad: Mittel **Vorkenntnisse**: Uebung_006a_AX (SR/T-Flip-Flop mit drei Tastern), Grundlagen der logiBUS-Tastenereignisse (logiBUS_IE).

Zusammenfassung

Die Übung `Uebung_006a5_AX` zeigt, wie der Alles-Könner-Baustein `AX_T_FF_SR` mit einem einzigen Taster bedient werden kann, indem drei `logiBUS_IE`-Bausteine dieselbe physische Eingangsquelle `Input_I1` auf unterschiedliche Gesten hin auswerten. Langer Druck setzt und dessen Loslassen setzt zurück, ein kurzer Klick toggelt – damit vereint ein Taster rastendes und tastendes Verhalten in einem Bauteil.

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

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

2.1.2.93 Uebung_006a_AX: SR und T-Flip-Flop mit 3x IE



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_006a_AX`. Diese Übung zeigt einen „Alles-Könner“-Baustein.

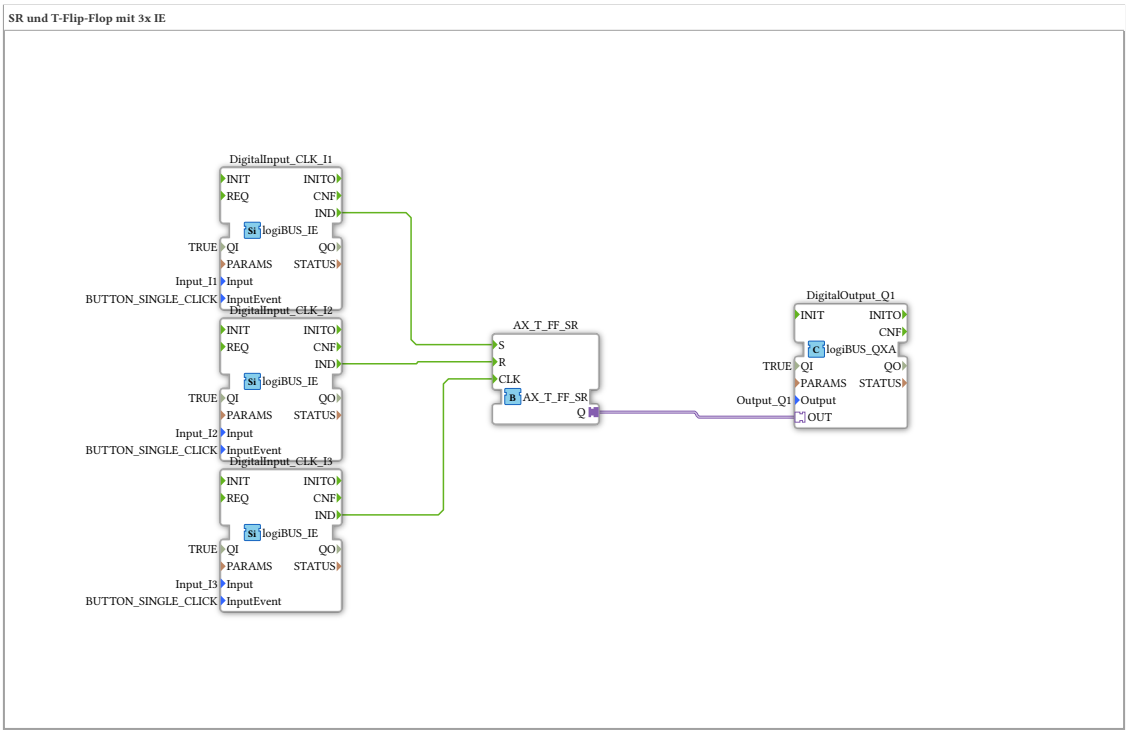
Ziel der Übung

Kennenlernen des `AX_T_FF_SR`.

Beschreibung und Komponenten

Die Subapplikation `Uebung_006a_AX.SUB` nutzt drei Taster.

Funktionsbausteine (FBs)



- **`I1` (Set)**
- **`I2` (Reset)**
- **`I3` (Toggle)**
- **`AX_T_FF_SR`**: Vereint Toggle, Set und Reset in einem Baustein.

Funktionsweise

- `I1` schaltet an.
- `I2` schaltet aus.

- `I3` schaltet um.

Dies bietet maximale Flexibilität für die Bedienung.

Anwendungsbeispiel

Smart Home Lichtsteuerung:

- Taster an der Wand: Toggle (`I3`).
- Zentral „Alles Aus“ beim Verlassen des Hauses: Reset (`I2`).
- „Panik-Licht“ (Alarmanlage): Set (`I1`).

2.1.2.94 Uebung_006b_AX: RS-Flip-Flop mit 2x IE

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_006b_AX`.

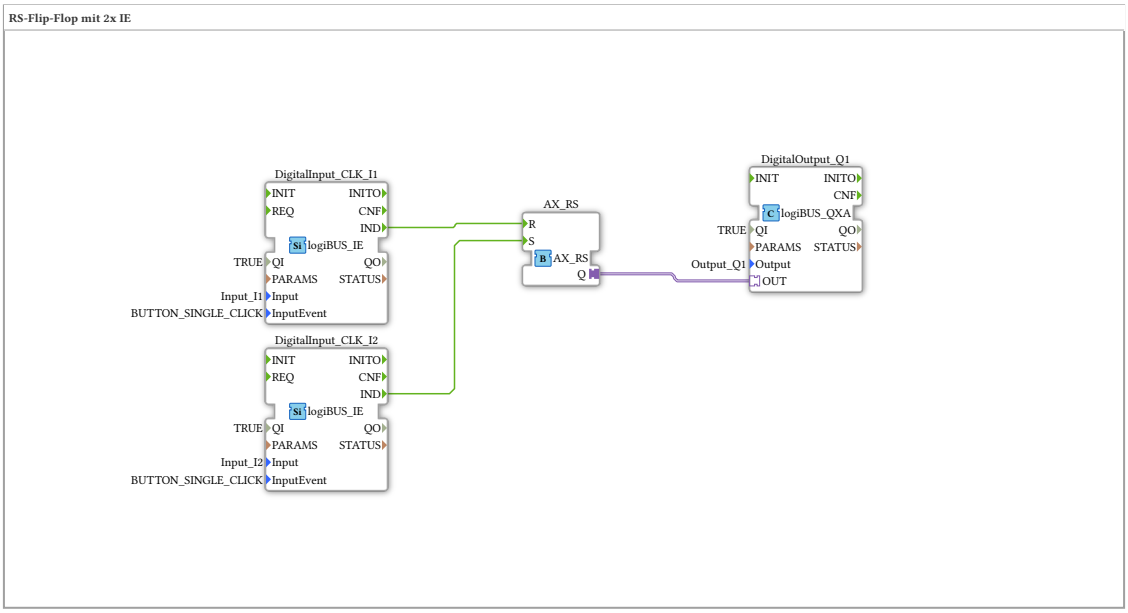
Ziel der Übung

Unterschied zwischen SR (Set Priority) und RS (Reset Priority) verstehen.

Beschreibung und Komponenten

Die Subapplikation `Uebung_006b_AX.SUB` nutzt einen `AX_RS` Baustein.

Funktionsbausteine (FBs)



- **`AX_RS`**: Ein RS-Flip-Flop.

Funktionsweise

Funktional sehr ähnlich zu `AX_SR`. Der Unterschied liegt im Verhalten, wenn **gleichzeitig** (im selben SPS-Zyklus) ein Set- und ein Reset-Event eintreffen (oder wenn beide Eingänge TRUE sind bei pegelgesteuerten Bausteinen).

- **SR**: Setzen hat Vorrang -> Ausgang wird TRUE.
- **RS**: Rücksetzen hat Vorrang -> Ausgang wird FALSE.

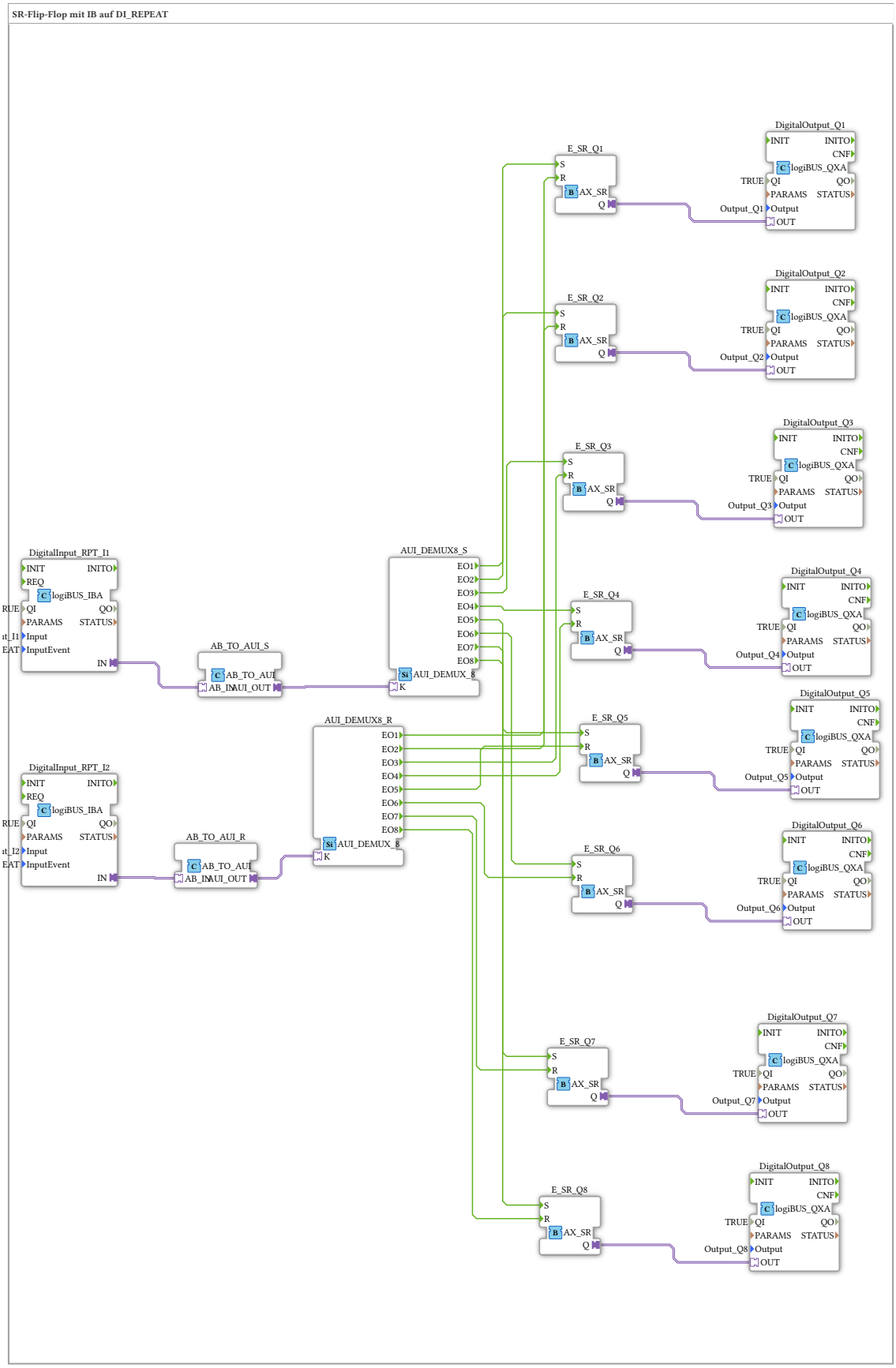
In der IEC 61499 mit Event-Verarbeitung ist „Gleichzeitigkeit“ subtiler, da Events oft sequenziell abgearbeitet werden. Wenn jedoch z.B. durch einen `E_SPLIT` beide Events im selben „Step“ ankommen, entscheidet die interne Logik des Bausteins. Beim `AX_RS` gewinnt im Zweifel das Reset.

Anwendungsbeispiel

Sicherheitskritische Abschaltung: Wenn jemand „Start“ drückt, während „Not-Aus“ gedrückt ist, darf die Maschine **nicht** anlaufen. Daher Reset-Dominanz (RS).

2.1.2.95 Uebung_006c_AX: SR-Flip-Flop mit IB auf DI_REPEAT

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_006c_AX (SR-Flip-Flop mit IB auf DI_REPEAT).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **SR-Flip-Flop mit IB auf DI_REPEAT**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_006c_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
- **DigitalInput_RPT_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IBA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
  - Parameter InputEvent = BUTTON_PRESS_REPEAT
- **DigitalInput_RPT_I2**: Instanz des Typs logiBUS::io::DI::logiBUS_IBA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I2
  - Parameter InputEvent = BUTTON_PRESS_REPEAT
- **E_SR_Q1**: Instanz des Typs adapter::events::unidirectional::AX_SR.
- **E_SR_Q2**: Instanz des Typs adapter::events::unidirectional::AX_SR.
- **DigitalOutput_Q2**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q2
- **DigitalOutput_Q3**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q3
- **DigitalOutput_Q4**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q4
- **E_SR_Q5**: Instanz des Typs adapter::events::unidirectional::AX_SR.
- **E_SR_Q6**: Instanz des Typs adapter::events::unidirectional::AX_SR.
- **DigitalOutput_Q5**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q5
- **DigitalOutput_Q6**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q6
- **DigitalOutput_Q7**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q7
- **DigitalOutput_Q8**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q8
- **AUI_DEMUX8_S**: Instanz des Typs adapter::events::unidirectional::AUI_DEMUX_8.
- **AUI_DEMUX8_R**: Instanz des Typs adapter::events::unidirectional::AUI_DEMUX_8.
- **E_SR_Q3**: Instanz des Typs adapter::events::unidirectional::AX_SR.
- **E_SR_Q4**: Instanz des Typs adapter::events::unidirectional::AX_SR.
- **E_SR_Q7**: Instanz des Typs adapter::events::unidirectional::AX_SR.
- **E_SR_Q8**: Instanz des Typs adapter::events::unidirectional::AX_SR.
- **AB_TO_AUI_S**: Instanz des Typs adapter::conversion::unidirectional::AB_TO_AUI.
- **AB_TO_AUI_R**: Instanz des Typs adapter::conversion::unidirectional::AB_TO_AUI.

Verbindungen und Schnittstellen

Adapterverbindungen:

- DigitalInput_RPT_I1.IN -> AB_TO_AUI_S.AB_IN
- AB_TO_AUI_S.AUI_OUT -> AUI_DEMUX8_S.K
- DigitalInput_RPT_I2.IN -> AB_TO_AUI_R.AB_IN
- AB_TO_AUI_R.AUI_OUT -> AUI_DEMUX8_R.K
- E_SR_Q1.Q -> DigitalOutput_Q1.OUT

- E_SR_Q2.Q -> DigitalOutput_Q2.OUT
- E_SR_Q3.Q -> DigitalOutput_Q3.OUT
- E_SR_Q4.Q -> DigitalOutput_Q4.OUT
- E_SR_Q5.Q -> DigitalOutput_Q5.OUT
- E_SR_Q6.Q -> DigitalOutput_Q6.OUT
- E_SR_Q7.Q -> DigitalOutput_Q7.OUT
- E_SR_Q8.Q -> DigitalOutput_Q8.OUT

Ereignisverbindungen:

- AUI_DEMUX8_S.EO1 -> E_SR_Q1.S
- AUI_DEMUX8_S.EO2 -> E_SR_Q2.S
- AUI_DEMUX8_S.EO3 -> E_SR_Q3.S
- AUI_DEMUX8_S.EO4 -> E_SR_Q4.S
- AUI_DEMUX8_S.EO5 -> E_SR_Q5.S
- AUI_DEMUX8_S.EO6 -> E_SR_Q6.S
- AUI_DEMUX8_S.EO7 -> E_SR_Q7.S
- AUI_DEMUX8_S.EO8 -> E_SR_Q8.S
- AUI_DEMUX8_R.EO1 -> E_SR_Q1.R
- AUI_DEMUX8_R.EO2 -> E_SR_Q2.R
- AUI_DEMUX8_R.EO3 -> E_SR_Q3.R
- AUI_DEMUX8_R.EO4 -> E_SR_Q4.R
- AUI_DEMUX8_R.EO5 -> E_SR_Q5.R
- AUI_DEMUX8_R.EO6 -> E_SR_Q6.R
- AUI_DEMUX8_R.EO7 -> E_SR_Q7.R
- AUI_DEMUX8_R.EO8 -> E_SR_Q8.R

Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_006c_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.96 Uebung_006d_AX: SR-Flip-Flop mit 2x IE und Doppelklick



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_006d_AX`.

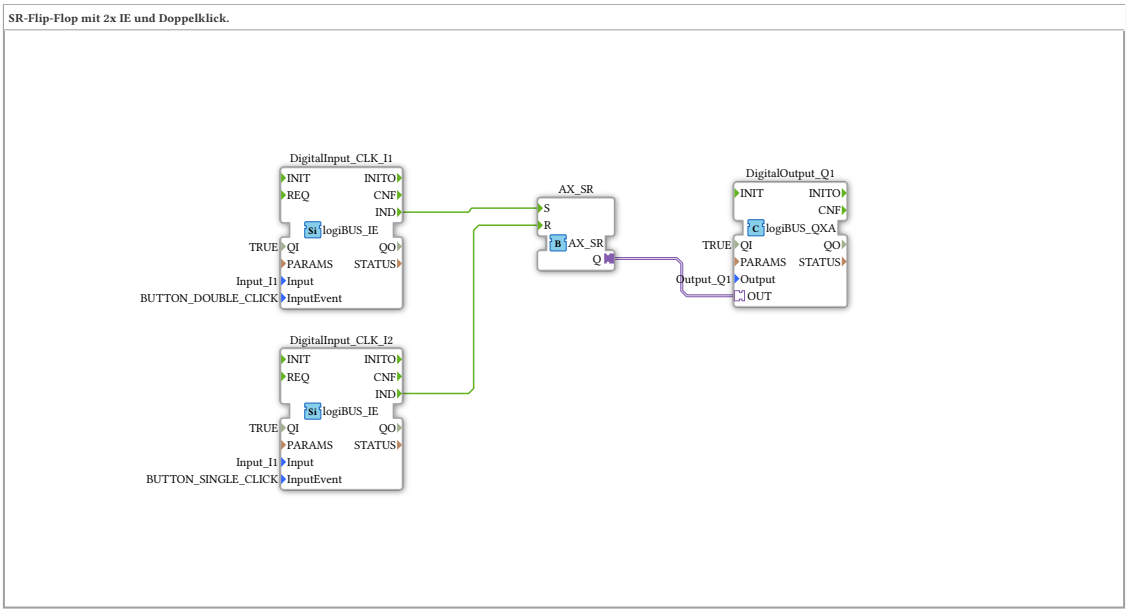
Ziel der Übung

Kombination von Input-Events und Speichergliedern.

Beschreibung und Komponenten

Die Subapplikation `Uebung_006d_AX.SUB` definiert eine asymmetrische Bedienung.

Funktionsbausteine (FBs)



- **`I1` (Set)**: Konfiguriert auf `BUTTON_DOUBLE_CLICK`.
- **`I2` (Reset)**: Konfiguriert auf `BUTTON_SINGLE_CLICK`.
- **`AX_SR`**: Speicher.

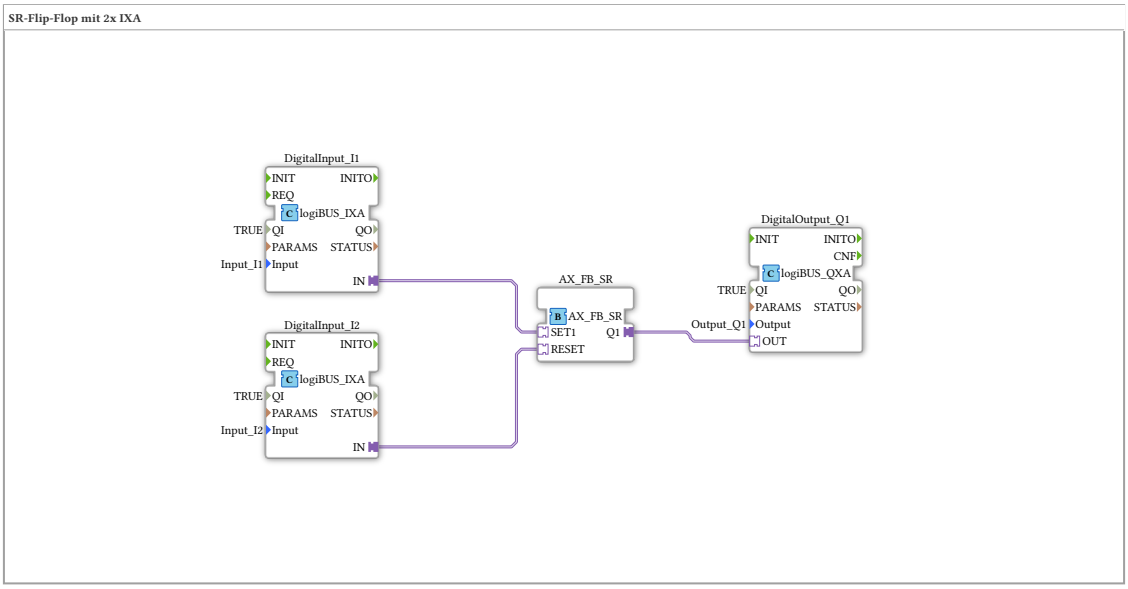
Funktionsweise

- Zum **Einschalten** muss man den Taster `I1` **doppelt** klicken.
- Zum **Ausschalten** reicht ein **einfacher** Klick auf `I2`.

Anwendungsbeispiel

Schutz vor versehentlichem Einschalten: Ein Gerät (z.B. eine Pumpe), das gefährlich sein kann oder viel Energie verbraucht, soll nicht durch ein versehentliches Berühren des Schalters starten. Der Doppelklick fordert eine bewusste Handlung („Ja, ich will wirklich“). Das Ausschalten muss hingegen im Notfall schnell und einfach gehen (Einzelklick).

2.1.2.97 Uebung_006e1_AX: SR-Flip-Flop mit 2x IXA



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung implementiert ein **SR-Flip-Flop** (bistabiles Kippglied) unter Verwendung von **Adapter-Technologie** (AX/IX/QX). Die Logik dient dazu, einen digitalen Ausgang über einen Eingang zu setzen (Set) und über einen zweiten Eingang zurückzusetzen (Reset). Durch die Verwendung von Adaptern werden Daten- und Ereignisflüsse in einzelnen Verbindungen gebündelt, was die Übersichtlichkeit im Plan erhöht.

Verwendete Funktionsbausteine (FBs)

In dieser Sub-Applikation werden spezifische Bausteine für die Ein- und Ausgabe über den logiBUS sowie ein Logikbaustein für das Flip-Flop verwendet.

Sub-Bausteine: Uebung_006e1_AX

Hier sind die internen Funktionsbausteine aufgeführt, die in diesem Netzwerk verschaltet sind:

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **DigitalInput_I1**: `logiBUS::io::DI::logiBUS_IXA`
    - **Parameter**:
      - `Input` = „Input_I1“
      - `QI` = TRUE (sichtbar: false)
    - **Beschreibung**: Dieser Baustein stellt den ersten digitalen Eingang dar (Setzen). Er nutzt einen Adapter-Ausgang (`IN`), um den Zustand an die Logik weiterzugeben.
  - **DigitalInput_I2**: `logiBUS::io::DI::logiBUS_IXA`
    - **Parameter**:
      - `Input` = „Input_I2“
      - `QI` = TRUE (sichtbar: false)
    - **Beschreibung**: Dieser Baustein stellt den zweiten digitalen Eingang dar (Rücksetzen). Er nutzt ebenfalls einen Adapter-Ausgang (`IN`).
  - **DigitalOutput_Q1**: `logiBUS::io::DQ::logiBUS_QXA`
    - **Parameter**:
      - `Output` = „Output_Q1“
      - `QI` = TRUE (sichtbar: false)
    - **Beschreibung**: Dieser Baustein repräsentiert den digitalen Ausgang. Er empfängt Signale über einen Adapter-Eingang (`OUT`).
  - **AX_FB_SR**: `adapter::iec61131::bistableElements::AX_FB_SR`
    - **Beschreibung**: Dies ist der Kern-Logikbaustein. Es handelt sich um ein SR-Flip-Flop, das speziell für die Verwendung mit Adaptern ausgelegt ist. Es besitzt Adapter-Eingänge für `SET1` und `RESET` sowie einen Adapter-Ausgang für `Q1`.

Programmablauf und Verbindungen

Das Netzwerk realisiert eine Speicherfunktion mittels eines SR-Flip-Flops. Der Ablauf und die Adapter-Verbindungen gestalten sich wie folgt:

1. **Setzen (Set):**
   - Der Adapter-Ausgang `IN` von **DigitalInput_I1** ist mit dem Adapter-Eingang `SET1` des **AX_FB_SR** Bausteins verbunden.
   - Wird `Input_I1` aktiv, wird das Flip-Flop gesetzt.
2. **Rücksetzen (Reset):**
   - Der Adapter-Ausgang `IN` von **DigitalInput_I2** ist mit dem Adapter-Eingang `RESET` des **AX_FB_SR** Bausteins verbunden.
   - Wird `Input_I2` aktiv, wird das Flip-Flop zurückgesetzt.
3. **Ausgabe:**

- Der Adapter-Ausgang `Q1` des **AX_FB_SR** Bausteins ist mit dem Adapter-Eingang `OUT` von **DigitalOutput_Q1** verbunden.
- Der Status des Flip-Flops wird direkt an den physischen Ausgang `Output_Q1` weitergeleitet.

Besonderheit der Adapter: Anstatt separate Ereignis- (Events) und Datenleitungen zu ziehen, werden hier Adapter-Verbindungen (dargestellt durch die Doppelpfeile/breiteren Linien in der IDE) genutzt. Dies reduziert die Anzahl der sichtbaren Leitungen drastisch.

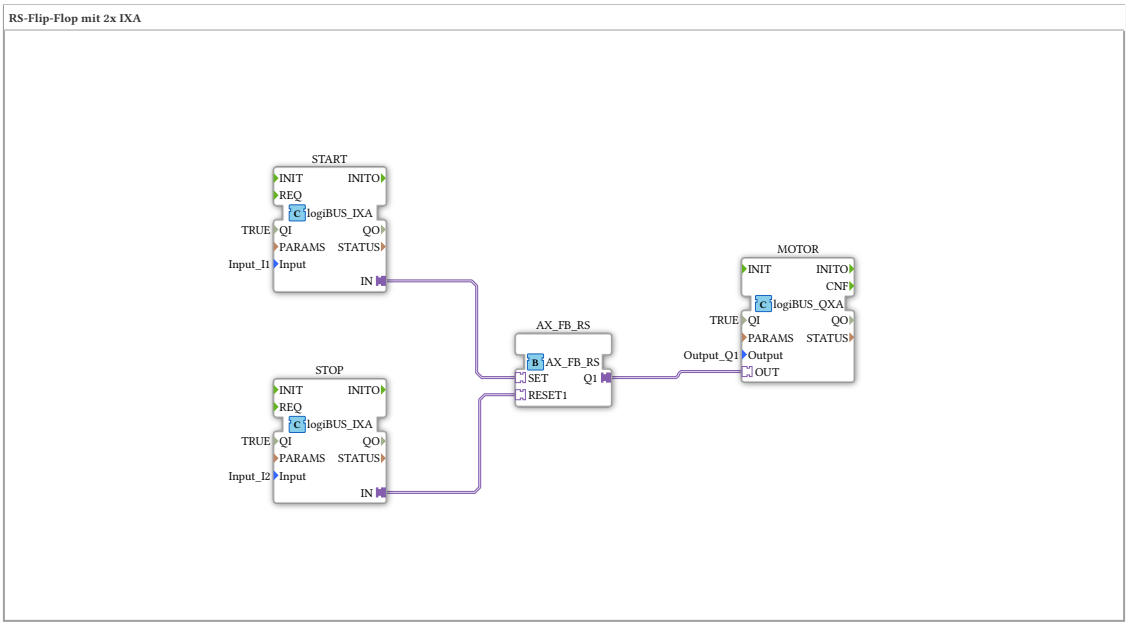
Logikverhalten (SR-Dominanz): Da es sich um einen SR-Baustein handelt, gilt typischerweise: Ist nur Setzen aktiv, ist der Ausgang

1. Ist nur Rücksetzen aktiv, ist der Ausgang
2. Sind beide Eingänge gleichzeitig aktiv, bestimmt der Bausteintyp die Dominanz (bei SR-Bausteinen nach IEC 61131 hat oft das Setzen Vorrang, dies ist jedoch implementationabhängig vom spezifischen `AX_FB_SR`).

Zusammenfassung

Die Übung `Uebung_006e1_AX` demonstriert effizient den Einsatz von Adapter-Technologie zur Realisierung einer klassischen Speicherfunktion (SR-Flip-Flop). Durch die Verwendung der `logiBUS_IXA` und `logiBUS_QXA` Bausteine wird die Verbindung zur Hardware abstrahiert, während der `AX_FB_SR` die logische Verknüpfung übernimmt.

2.1.2.98 Uebung_006e2_AX: RS-Flip-Flop mit 2x IXA



• – ○ • – ○ • – ○ •

Einleitung

Die Übung **Uebung_006e2_AX** realisiert ein RS-Flip-Flop (Rücksetz-Dominant) unter Verwendung von Adapter-Verbindungen (AX). Ziel der Übung ist es, zwei digitale Eingänge zu nutzen, um einen digitalen Ausgang zu setzen (Set) oder zurückzusetzen (Reset). Dabei wird die Logikbaustein-Bibliothek für bistabile Elemente verwendet und über das logiBUS-System mit der Hardware abstrahiert.

Verwendete Funktionsbausteine (FBs)

In dieser Sub-Applikation werden spezifische Funktionsbausteine für die Ein- und Ausgabe sowie die logische Verarbeitung verwendet.

Sub-Bausteine: Uebung_006e2_AX

Diese Übung selbst ist als `SubAppType` definiert und enthält folgende interne Komponenten:

- **Typ**: SubAppType
- **Verwendete interne FBs**:

  - **DigitalInput_I1**: `logiBUS::io::DI::logiBUS_IXA`
    - Parameter: `Input` = „Input_I1“
    - Parameter: `QI` = „TRUE“
    - Beschreibung: Adapter-Baustein für den ersten digitalen Eingang.
  - **DigitalInput_I2**: `logiBUS::io::DI::logiBUS_IXA`
    - Parameter: `Input` = „Input_I2“
    - Parameter: `QI` = „TRUE“
    - Beschreibung: Adapter-Baustein für den zweiten digitalen Eingang.
  - **DigitalOutput_Q1**: `logiBUS::io::DQ::logiBUS_QXA`
    - Parameter: `Output` = „Output_Q1“
    - Parameter: `QI` = „TRUE“
    - Beschreibung: Adapter-Baustein für den digitalen Ausgang.
  - **AX_FB_RS**: `adapter::iec61131::bistableElements::AX_FB_RS`
    - Beschreibung: Ein bistabiles Element (RS-Flip-Flop) mit Adapter-Schnittstellen. Es realisiert die Speicherfunktion.
- **Funktionsweise**: Die Sub-Applikation liest zwei externe Signale über die logiBUS-Adapter ein, verarbeitet diese in einem RS-Flip-Flop und gibt den resultierenden Zustand an einen Ausgangsadapter weiter.

Programmablauf und Verbindungen

Der Programmablauf wird durch Adapter-Verbindungen (`AdapterConnections`) realisiert, die sowohl Daten als auch Ereignisse kapseln.

1. **Setzen des Speichers (Set):**
   - Der Adapter `DigitalInput_I1.IN` ist mit dem Adapter-Eingang `AX_FB_RS.SET` verbunden.
   - Wenn der Eingang `Input_I1` aktiviert wird, wird das RS-Flip-Flop gesetzt.
2. **Rücksetzen des Speichers (Reset):**
   - Der Adapter `DigitalInput_I2.IN` ist mit dem Adapter-Eingang `AX_FB_RS.RESET1` verbunden.
   - Wenn der Eingang `Input_I2` aktiviert wird, wird das RS-Flip-Flop zurückgesetzt.
3. **Ausgabe des Zustands:**
   - Der Adapter-Ausgang `AX_FB_RS.Q1` ist mit dem Adapter `DigitalOutput_Q1.OUT` verbunden.
   - Der aktuelle Zustand des Flip-Flops wird somit direkt an den physischen Ausgang `Output_Q1` weitergeleitet.

Lernziele:

- Verständnis von bistabilen Elementen (RS-Flip-Flop).
- Umgang mit Adapter-Verbindungen (AX/IXA/QXA) in 4diac zur Vereinfachung des Signalflusses.
- Verknüpfung von Hardware-IOs mit logischen Funktionen.

Zusammenfassung

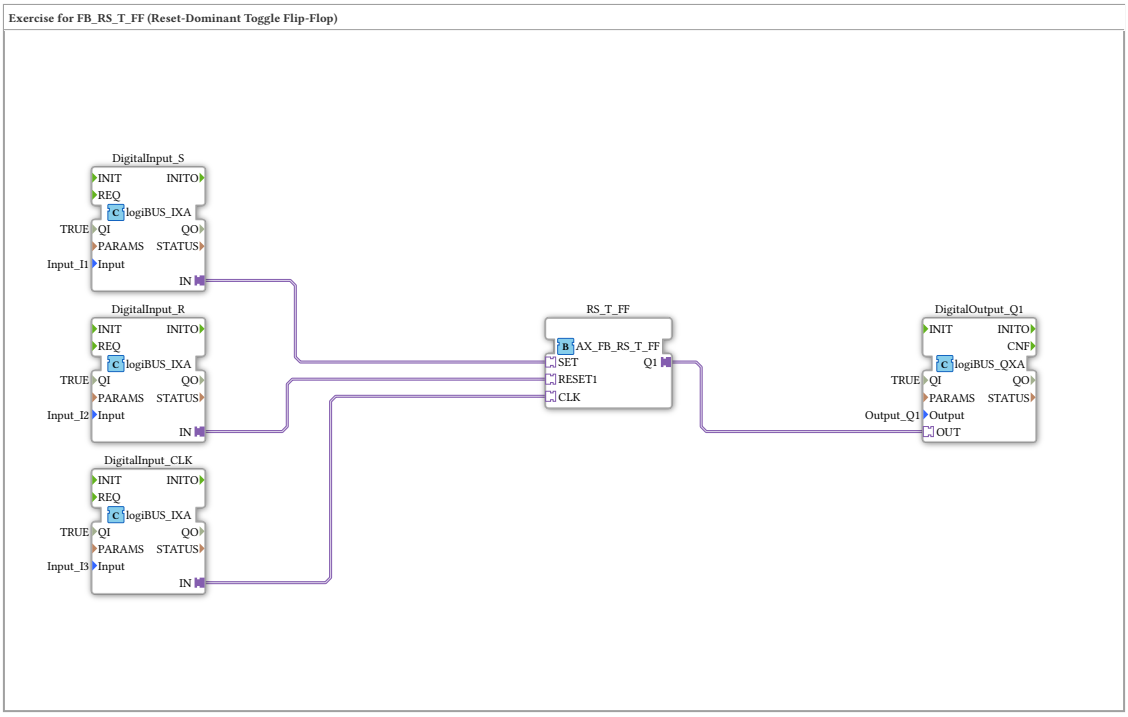
Die `Uebung_006e2_AX` demonstriert eine grundlegende Speicherfunktion in der Steuerungstechnik. Durch die Verwendung von Adaptern (`AX_FB_RS`, `logiBUS_IXA`, `logiBUS_QXA`) wird der Schaltplan übersichtlich gehalten, da Ereignis- und Datenflüsse in einer einzigen Verbindungslinie zusammengefasst sind. Das Verhalten entspricht einem klassischen RS-Flip-Flop, bei dem `Input_I1` als Set-Eingang und `Input_I2` als Reset-Eingang fungiert.

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

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

2.1.2.99 Uebung_006f_RS_AX: Exercise for FB_RS_T_FF (Reset-Dominant Toggle Flip-Flop)

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_006f_RS_AX (Exercise for FB_RS_T_FF (Reset-Dominant Toggle Flip-Flop)).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Exercise for FB_RS_T_FF (Reset-Dominant Toggle Flip-Flop)**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_006f_RS_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_S**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
- **DigitalInput_R**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I2
- **DigitalInput_CLK**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I3
- **RS_T_FF**: Instanz des Typs adapter::bistableElements::AX_FB_RS_T_FF.
- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1

Verbindungen und Schnittstellen

Adapterverbindungen:

- DigitalInput_S.IN -> RS_T_FF.SET
- DigitalInput_R.IN -> RS_T_FF.RESET1
- DigitalInput_CLK.IN -> RS_T_FF.CLK
- RS_T_FF.Q1 -> DigitalOutput_Q1.OUT

Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.

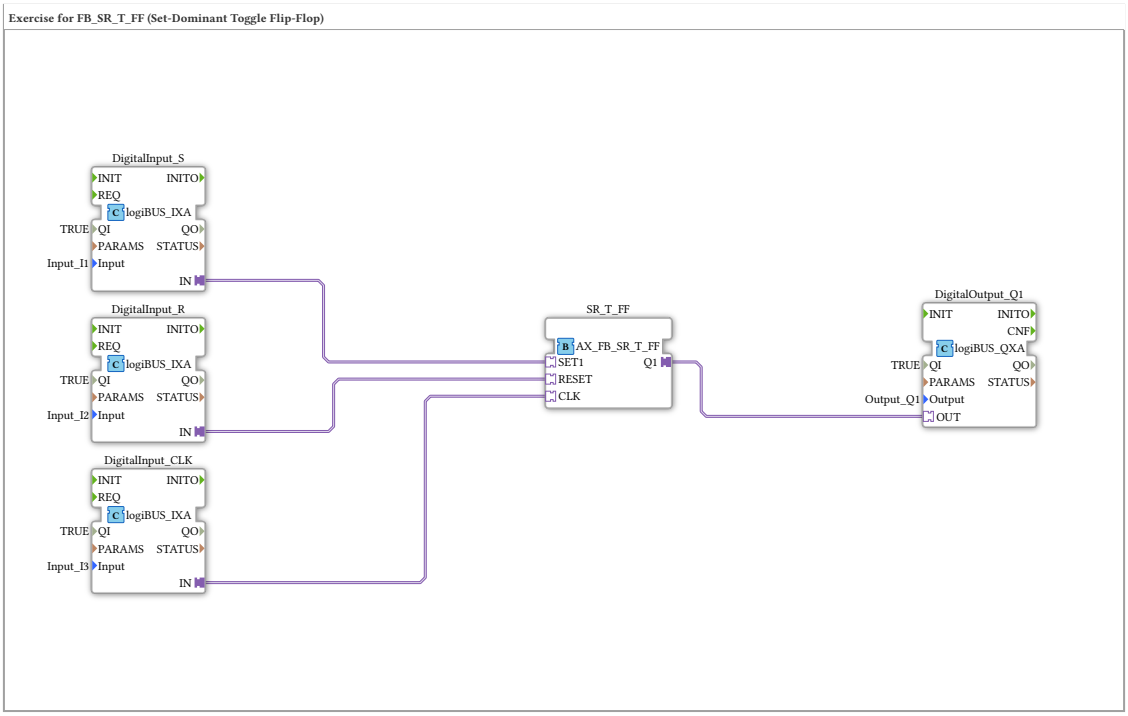
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_006f_RS_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.100 Uebung_006f_SR_AX: Exercise for FB_SR_T_FF (Set-Dominant Toggle Flip-Flop)

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_006f_SR_AX (Exercise for FB_SR_T_FF (Set-Dominant Toggle Flip-Flop)).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Exercise for FB_SR_T_FF (Set-Dominant Toggle Flip-Flop)**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_006f_SR_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_S**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
- **DigitalInput_R**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I2
- **DigitalInput_CLK**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I3
- **SR_T_FF**: Instanz des Typs adapter::bistableElements::AX_FB_SR_T_FF.

- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1

Verbindungen und Schnittstellen

Adapterverbindungen:

- DigitalInput_S.IN -> SR_T_FF.SET1
- DigitalInput_R.IN -> SR_T_FF.RESET
- DigitalInput_CLK.IN -> SR_T_FF.CLK
- SR_T_FF.Q1 -> DigitalOutput_Q1.OUT

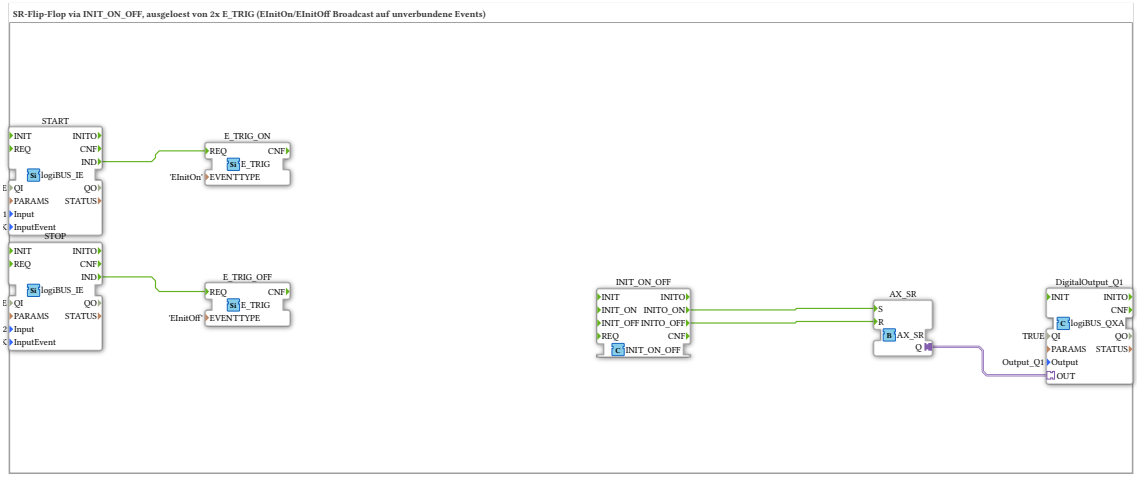
Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_006f_SR_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.101 Uebung_006g_AX: SR-Flip-Flop via INIT_ON_OFF, ausgelöst von zwei E_TRIG



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung setzt ein SR-Flip-Flop (`AX_SR`) nicht über direkte Event-Leitungen, sondern über das Broadcast-Verhalten von `E_TRIG` an: Zwei `E_TRIG`-Bausteine lösen die unverbundenen Ereignis-Eingänge `INIT_ON` bzw. `INIT_OFF` des Bausteins `INIT_ON_OFF` aus. Dessen durchgereichte Ausgänge `INITO_ON` und `INITO_OFF` setzen und rücksetzen das Flip-Flop, dessen Zustand den Ausgang `Q1` steuert. Ein Klick auf `I1` schaltet `Q1` ein, ein Klick auf `I2` schaltet `Q1` aus.

Verwendete Funktionsbausteine (FBs)

- **START**: logiBUS Eingang mit Klick-Erkennung für den Start-Taster
  - **Typ**: `logiBUS::io::DI::logiBUS_IE`
  - **Parameter**: `QI = TRUE`, `Input = Input_I1`, `InputEvent = BUTTON_SINGLE_CLICK`
  - **Erklärung**: Erkennt einen einzelnen Klick auf `I1` und meldet ihn als Event `IND`.
- **STOP**: logiBUS Eingang mit Klick-Erkennung für den Stopp-Taster
  - **Typ**: `logiBUS::io::DI::logiBUS_IE`
  - **Parameter**: `QI = TRUE`, `Input = Input_I2`, `InputEvent = BUTTON_SINGLE_CLICK`

  - **Erklärung**: Erkennt einen einzelnen Klick auf `I2` und meldet ihn als Event `IND`.
- **E_TRIG_ON**: Auslöser für den Ereignistyp `EInitOn`
  - **Typ**: `iec61499::events::E_TRIG`
  - **Parameter**: `EVENTTYPE = 'EInitOn'`
  - **Erklärung**: Löst bei `REQ` alle unverbundenen Ereignis-Eingänge mit dem Namen `EInitOn` im Netzwerk aus.
- **E_TRIG_OFF**: Auslöser für den Ereignistyp `EInitOff`
  - **Typ**: `iec61499::events::E_TRIG`
  - **Parameter**: `EVENTTYPE = 'EInitOff'`
  - **Erklärung**: Löst bei `REQ` alle unverbundenen Ereignis-Eingänge mit dem Namen `EInitOff` im Netzwerk aus.
- **INIT_ON_OFF**: Durchreicher für die beiden Init-Kanäle
  - **Typ**: `iec61131::booleanOperators::INIT_ON_OFF`
  - **Parameter**: keine
  - **Erklärung**: Besitzt neben `INIT` und `REQ` zwei eigenständige Pass-Through-Kanäle: `INIT_ON` → `INITO_ON` (Ereignistyp `EInitOn`) und `INIT_OFF` → `INITO_OFF` (Ereignistyp `EInitOff`). Die Eingänge `INIT_ON` und `INIT_OFF` bleiben in dieser Übung bewusst unverbunden (siehe Hinweis im Modell).
- **AX_SR**: Set-Reset-Flip-Flop mit Adapter-Ausgang
  - **Typ**: `adapter::events::unidirectional::AX_SR`
  - **Parameter**: keine
  - **Erklärung**: Wird über die Events `S` (Set) und `R` (Reset) gesteuert und stellt den Zustand am Adapter `Q` bereit.
- **DigitalOutput_Q1**: logiBUS Digitalausgang
  - **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
  - **Parameter**: `QI = TRUE`, `Output = Output_Q1`
  - **Erklärung**: Gibt den Zustand von `AX_SR.Q` an den physischen Ausgang `Q1` weiter.

Programmablauf und Verbindungen

1. `START.IND` → `E_TRIG_ON.REQ` (EventConnection): ein Klick auf `I1` löst den Trigger für `EInitOn` aus.
2. `STOP.IND` → `E_TRIG_OFF.REQ` (EventConnection): ein Klick auf `I2` löst den Trigger für `EInitOff` aus.
3. `INIT_ON_OFF.INITO_ON` → `AX_SR.S` und `INIT_ON_OFF.INITO_OFF` → `AX_SR.R` (EventConnections): die durchgereichten Ereignisse setzen bzw. rücksetzen das Flip-Flop.
4. `AX_SR.Q` → `DigitalOutput_Q1.OUT` (AdapterConnection): der Flip-Flop-Zustand wird auf den physischen Ausgang `Q1` gelegt.

Hinweis zum Broadcast: `INIT_ON` und `INIT_OFF` von `INIT_ON_OFF` sind nicht verbunden. `E_TRIG` sendet seinen `EVENTTYPE` an alle unverbundenen Ereignis-Eingänge dieses Typs im Netzwerk – daher reagiert hier nur der jeweils passende Kanal (`EInitOn` bzw. `EInitOff`).

Verhalten: Klick auf `I1` → `E_TRIG_ON` → `INIT_ON_OFF.INITO_ON` → `AX_SR.S`: `Q1` wird eingeschaltet. Klick auf `I2` → `E_TRIG_OFF` → `INIT_ON_OFF.INITO_OFF` → `AX_SR.R`: `Q1` wird ausgeschaltet.

Zusammenfassung

Die Übung `Uebung_006g_AX` zeigt, dass ein SR-Flip-Flop auch ohne direkte Event-Leitungen zwischen Taster und Flip-Flop geschaltet werden kann: `E_TRIG` broadcastet einen Ereignistyp an unverbundene Ereignis-Eingänge, `INIT_ON_OFF` reicht die beiden Kanäle `EInitOn` und `EInitOff` getrennt durch und `AX_SR` setzt bzw. rücksetzt daraus den Ausgang `Q1`.

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

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

2.1.2.102 Uebung_007_AX: Blinker mit E_CYCLE und E_T_FF



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_007_AX`. Hier wird gezeigt, wie man zeitgesteuerte Ereignisse erzeugt.

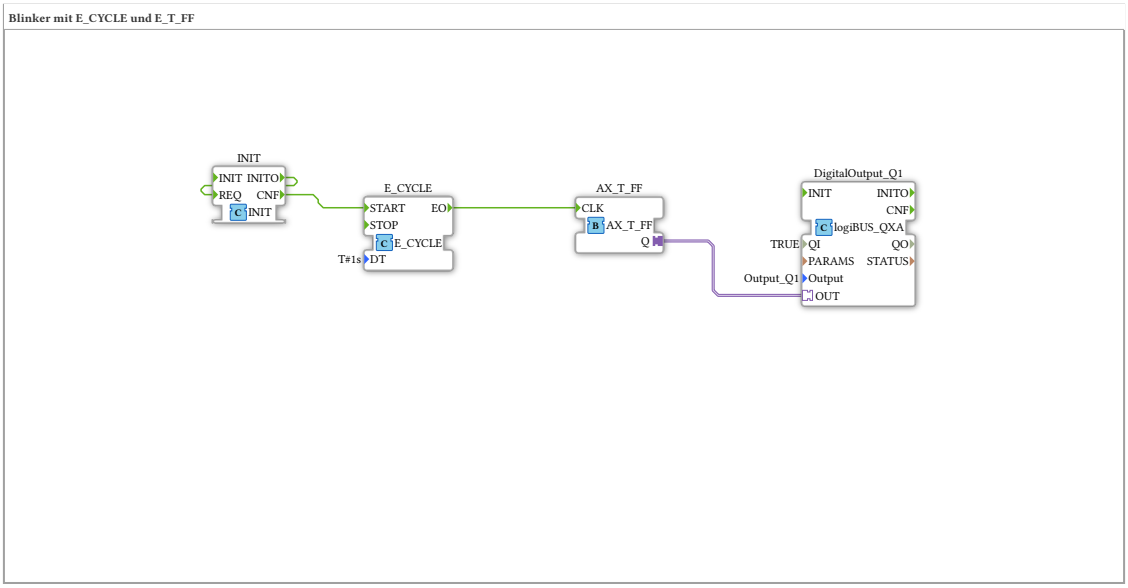
Ziel der Übung

Erzeugung eines periodischen Blinksignals.

Beschreibung und Komponenten

Die Subapplikation `Uebung_007_AX.SUB` nutzt einen `E_CYCLE` Baustein in Kombination mit einem Flip-Flop.

Funktionsbausteine (FBs)



- **`E_CYCLE`**: Ein Ereignis-Generator. Er sendet periodisch Events am Ausgang `EO`. Der Parameter `DT` bestimmt die Periodendauer (hier `T#1s`).
- **`AX_T_FF`**: Das Toggle-Flip-Flop.
- **`DigitalOutput_Q1`**: Die Lampe.

Funktionsweise

1. Der `E_CYCLE` Baustein feuert jede Sekunde ein Event.
2. Das Event erreicht das Flip-Flop (`AX_T_FF.CLK`).
3. Das Flip-Flop schaltet um (An -> Aus -> An ...).
4. Die Lampe blinkt mit einer Frequenz von 0,5 Hz (1 Sekunde an, 1 Sekunde aus).

Anwendungsbeispiel

Warnleuchte: Eine Signallampe soll blinken, um Aufmerksamkeit zu erregen.

2.1.2.103 Uebung_007a1_AX: Blinker mit E_CYCLE und E_T_FF

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_007a1_AX`.

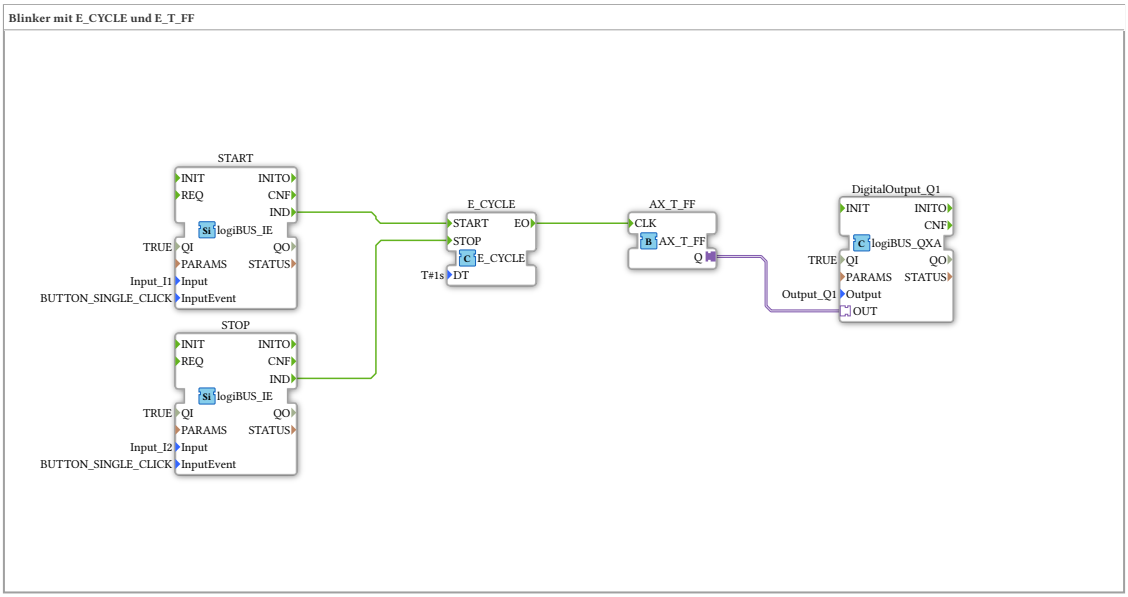
Ziel der Übung

Starten und Stoppen des Blinkers.

Beschreibung und Komponenten

Die Subapplikation `Uebung_007a1_AX.SUB` nutzt die Eingänge `START` und `STOP` des `E_CYCLE` Bausteins.

Funktionsbausteine (FBs)



- **`START` (I1)**: Startet den Zyklus.
- **`STOP` (I2)**: Stoppt den Zyklus.
- **`E_CYCLE`**: Generiert Events nur, wenn er gestartet ist.

Problem

Wie im Kommentar der Subapplikation vermerkt („dieser Blinker bleibt zufällig auf AN oder AUS stehen“): Wenn man `STOP` drückt, hört der `E_CYCLE` auf, Events zu senden. Das Flip-Flop `AX_T_FF` behält aber seinen *letzten* Zustand bei. War die Lampe gerade an, bleibt sie dauerhaft an. Das ist meistens nicht gewünscht (eine gestoppte Warnleuchte sollte aus sein).

Anwendungsbeispiel

Zeigt, warum man beim Design von Zustandsautomaten den „Stopp-Zustand“ definieren muss.

2.1.2.104 Uebung_007a2_AX: Blinker mit E_CYCLE und E_T_FF

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_007a2_AX`.

Ziel der Übung

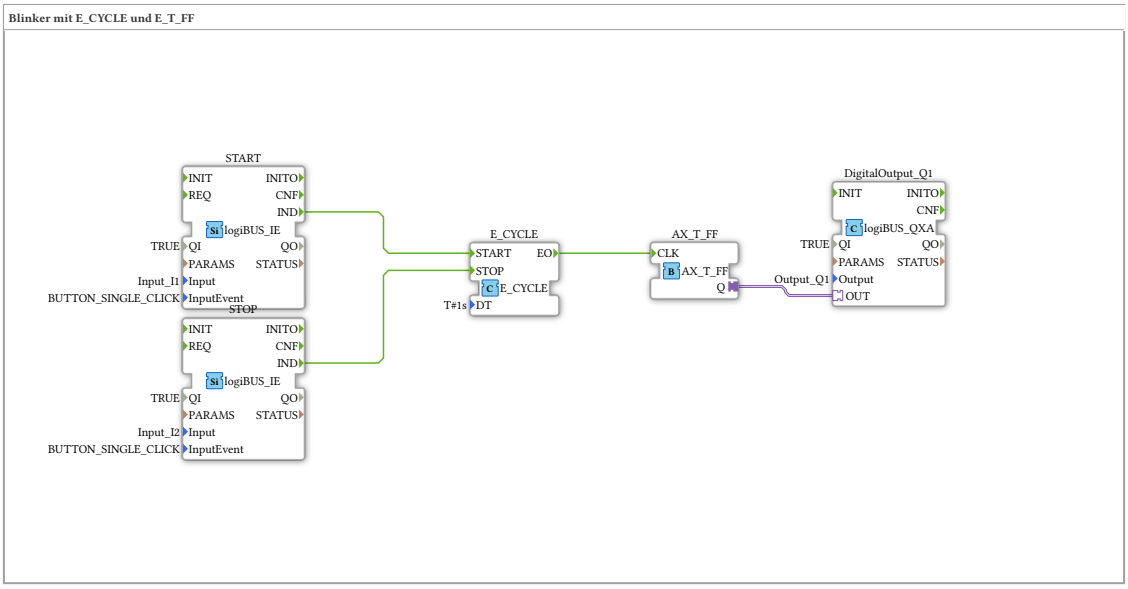
Untersuchung des Verhaltens.

Beschreibung

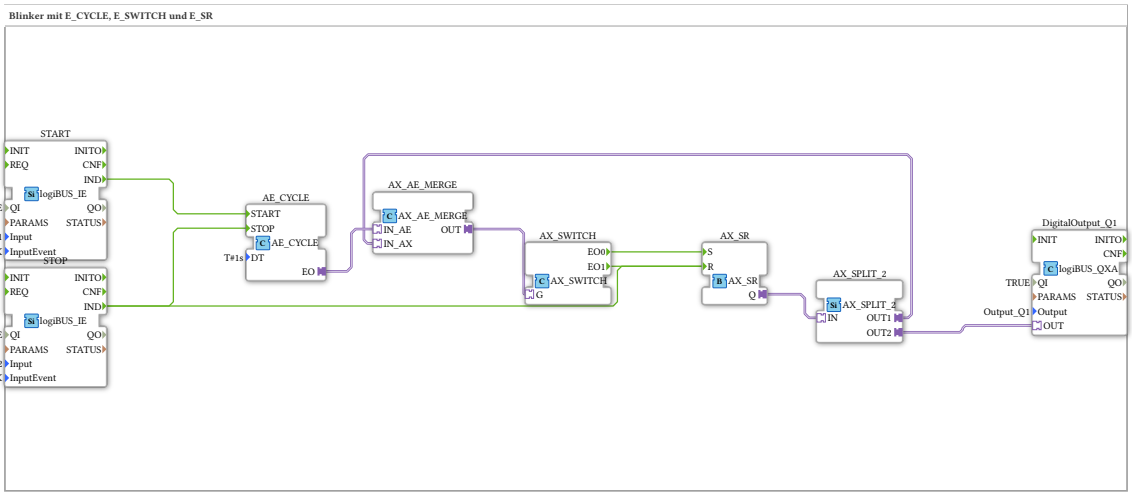
Strukturell sehr ähnlich zu `Uebung_007a1_AX`. Auch hier besteht das Problem des undefinierten Endzustands der Lampe. Es dient vermutlich als Wiederholung oder Variation im Layout.

Fazit

Auch diese Lösung ist für sicherheitskritische Blinker ungeeignet.



2.1.2.105 Uebung_007a3_AE: Blinker mit E_CYCLE, E_SWITCH und E_SR



• — ○ • — ○ • — ○ •

Einleitung

In dieser Übung wird ein Blinker realisiert, der einen digitalen Ausgang (Output_Q1) periodisch ein- und ausschaltet. Die Ansteuerung erfolgt über zwei Taster (Start/Stop). Als zentrale Elemente werden die Funktionsbausteine `AE_CYCLE` (Timer), `AX_SWITCH` (Umschalter), `AX_SR` (Setz-Rücksetz-Flipflop) sowie weitere Adapterbausteine verwendet. Die Besonderheit dieser Schaltung: Der Ausgang bleibt im ausgeschalteten Zustand definitiv aus – es kommt zu keinem ungewollten Einschalten.

Verwendete Funktionsbausteine (FBs)

- **DigitalOutput_Q1** (Typ: `logiBUS::io::DQ::logiBUS_QXA`)
  - Parameter: `QI` = TRUE, `Output` = „Output_Q1“
  - Adapterbaustein zur Ansteuerung eines physischen digitalen Ausgangs.
- **AE_CYCLE** (Typ: `adapter::events::unidirectional::timers::AE_CYCLE`)
  - Parameter: `DT` = T#1s (Periodendauer 1 Sekunde)
  - Zyklischer Timer, der nach einem Start-Ereignis in regelmäßigen Abständen ein Ereignis ausgibt.
- **START** (Typ: `logiBUS::io::DI::logiBUS_IE`)
  - Parameter: `QI` = TRUE, `Input` = „Input_I1“, `InputEvent` = „BUTTON_SINGLE_CLICK“

  - Eingangsbaustein für einen Taster. Bei einem einfachen Tastendruck wird ein Ereignis `IND` ausgelöst.
- **STOP** (Typ: `logiBUS::io::DI::logiBUS_IE`)
  - Parameter: `QI` = TRUE, `Input` = „Input_I2“, `InputEvent` = „BUTTON_SINGLE_CLICK“
  - Gleicher Typ wie START, dient zum Stoppen des Timers und Rücksetzen des Flipflops.
- **AX_SR** (Typ: `adapter::events::unidirectional::AX_SR`)
  - Ereignisgesteuertes Setz-/Rücksetz-Flipflop. Die Eingänge `S` (Set) und `R` (Reset) werden über Ereignisse aktiviert; der Ausgang `Q` liefert ein Adaptersignal.
- **AX_SWITCH** (Typ: `adapter::events::unidirectional::AX_SWITCH`)
  - Ereignisgesteuerter Umschalter. Abhängig vom Wert des Eingangs `G` wird das eingehende Signal entweder auf den Ausgang `EO0` oder `EO1` weitergeleitet.
- **AX_AE_MERGE** (Typ: `adapter::events::unidirectional::AX_AE_MERGE`)
  - Vereinigt einen Adaptereingang (`IN_AX`) und einen Ereigniseingang (`IN_AE`) zu einem gemeinsamen Ausgangssignal (`OUT`).
- **AX_SPLIT_2** (Typ: `adapter::events::unidirectional::AX_SPLIT_2`)
  - Verteilt ein eingehendes Adaptersignal auf zwei identische Ausgänge (`OUT1` und `OUT2`).

Programmablauf und Verbindungen

1. **Start** – Ein Tastendruck an Input_I1 (START) erzeugt ein Ereignis `IND`, das an den `START`-Eingang von `AE_CYCLE` geleitet wird. Der Timer beginnt zu laufen.
2. **Stop** – Ein Tastendruck an Input_I2 (STOP) erzeugt ein Ereignis `IND`, das sowohl an den `STOP`-Eingang von `AE_CYCLE` (Timer stoppt) als auch an den `R`-Eingang von `AX_SR` (Flipflop wird zurückgesetzt) geführt wird.
3. **Zyklus** – Der Timer `AE_CYCLE` erzeugt jede Sekunde ein Ereignis an seinem Ausgang `EO`. Dieses Ereignis wird mit dem Adaptersignal von `AX_SR` (`Q`) über `AX_AE_MERGE` zusammengeführt und an den `G`-Eingang von `AX_SWITCH` gesendet.
4. **Umschaltung** – `AX_SWITCH` leitet das ankommende Signal (vom Merge) abhängig vom Pegel an `G` entweder auf `EO0` (verbunden mit `S` von `AX_SR`) oder auf `EO1` (verbunden mit `R` von `AX_SR`). Dadurch wird bei jedem Timer-Impuls der Zustand des Flipflops umgeschaltet.
5. **Ausgabe** – Der Ausgang `Q` von `AX_SR` wird über `AX_SPLIT_2` auf zwei Wege verteilt:
   - `OUT1` geht zurück zu `AX_AE_MERGE` (über `IN_AX`), um die Rückkopplung zu schließen.
   - `OUT2` wird zum Eingang `OUT` von `DigitalOutput_Q1` geführt und schaltet den physischen Ausgang (Output_Q1).

Lernziele

- Verständnis ereignisgesteuerter Ablaufsteuerung mit Timer, Flipflop und Umschalter.
- Umgang mit Adapter-Bausteinen in der 4diac-IDE.

Schwierigkeitsgrad: Mittel **Vorkenntnisse**: Grundlagen der 4diac-IDE und ereignisgesteuerte Funktionsbausteine. **Start**: Nach dem Laden der SubApp in ein Projekt kann sie durch Zuweisung der Ein-/Ausgänge an die Hardware getestet werden.

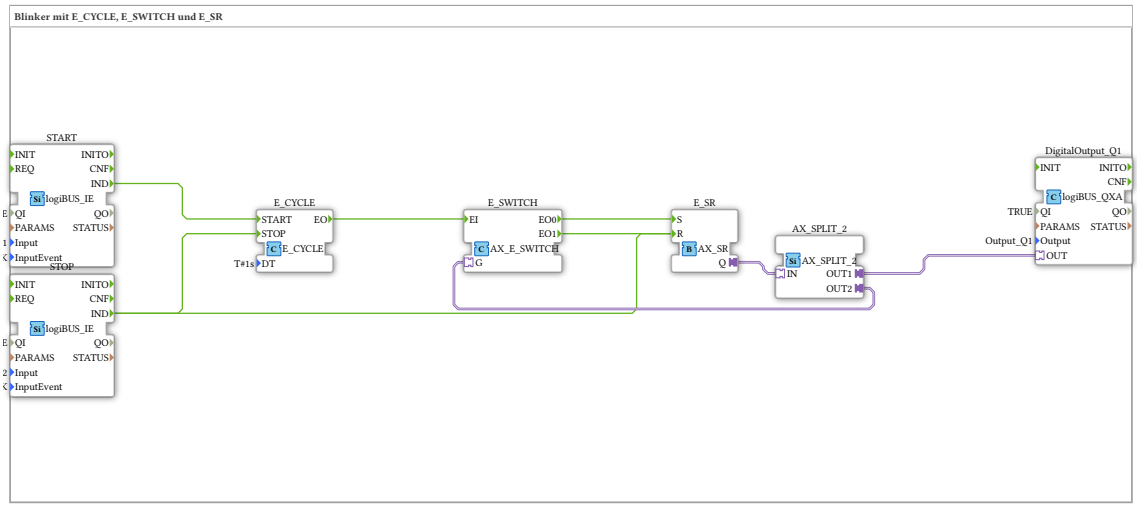
Zusammenfassung

Die Übung **Uebung_007a3_AE** demonstriert einen robusten Blinker, der durch die Kombination von zyklischem Timer, Umschalter und Setz-/Rücksetz-Flipflop realisiert wird. Durch die spezielle Verschaltung wird sichergestellt, dass der Ausgang nach einem Stopp zuverlässig ausgeschaltet bleibt. Der Aufbau eignet sich hervorragend zur Einführung in ereignisgesteuerte Logik mit Adaptern und zeigt, wie aus einfachen Grundbausteinen ein funktionales Steuerungsprogramm entsteht.

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

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

2.1.2.106 Uebung_007a3_AX: Blinker mit E_CYCLE, E_SWITCH und E_SR



• – ○ • – ○ • – ○ •

Einleitung

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_007a3_AX`. Diese Übung zeigt die „saubere“ Lösung für einen schaltbaren Blinker. Ziel ist, dass der Blinker nach dem Stoppen immer im Zustand „AUS“ stehen bleibt (Vorteil laut Kommentar im Netzwerk: „dieser Blinker bleibt immer auf AUS stehen“).

Verwendete Funktionsbausteine (FBs)

- **START**: logiBUS Eingang mit Klick-Erkennung für den Start
  - **Typ**: `logiBUS::io::DI::logiBUS_IE`
  - **Parameter**: `QI = TRUE`, `Input = Input_I1`, `InputEvent = BUTTON_SINGLE_CLICK`
  - **Erklärung**: Meldet einen Einzelklick auf `I1` als Event `IND`.
- **STOP**: logiBUS Eingang mit Klick-Erkennung für den Stopp
  - **Typ**: `logiBUS::io::DI::logiBUS_IE`
  - **Parameter**: `QI = TRUE`, `Input = Input_I2`, `InputEvent = BUTTON_SINGLE_CLICK`
  - **Erklärung**: Meldet einen Einzelklick auf `I2` als Event `IND`.
- **E_CYCLE**: Taktgeber
  - **Typ**: `iec61499::events::E_CYCLE`
  - **Parameter**: `DT = T#1s`
  - **Erklärung**: Erzeugt nach `START` im Abstand von einer Sekunde Ereignisse `EO`, bis `STOP` eintrifft.
- **E_SWITCH**: Ereignisweiche
  - **Typ**: `adapter::events::unidirectional::AX_E_SWITCH`
  - **Parameter**: keine
  - **Erklärung**: Leitet jedes Taktereignis `EI` abhängig vom über den Adapter `G` gelieferten Zustand entweder an `EO0` oder an `EO1` weiter.
- **E_SR**: Speicher („Blinker ist an“)
  - **Typ**: `adapter::events::unidirectional::AX_SR`
  - **Parameter**: keine
  - **Erklärung**: Set-Reset-Speicher mit Eingängen `S` und `R` und Ausgang `Q` (Adapter).
- **AX_SPLIT_2**: Signalverteiler
  - **Typ**: `adapter::events::unidirectional::AX_SPLIT_2`
  - **Parameter**: keine
  - **Erklärung**: Verteilt den Zustand von `E_SR.Q` an die Lampe (`OUT1`) und an die Rückkopplung zu `E_SWITCH.G` (`OUT2`).
- **DigitalOutput_Q1**: logiBUS Digitalausgang (die Lampe)
  - **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
  - **Parameter**: `QI = TRUE`, `Output = Output_Q1`

– **Erklärung**: Gibt den Zustand des Speichers an den physischen Ausgang `Q1` weiter.

Programmablauf und Verbindungen

1. `START.IND` ⟶ `E_CYCLE.START`: der Klick auf `I1` startet den Taktgeber.
2. `STOP.IND` ⟶ `E_CYCLE.STOP`: der Klick auf `I2` stoppt den Taktgeber.
3. `E_CYCLE.EO` ⟶ `E_SWITCH.EI`: jeder Takt läuft in die Ereignisweiche.
4. `E_SWITCH.EO0` ⟶ `E_SR.S` und `E_SWITCH.EO1` ⟶ `E_SR.R`: je nach Weichenstellung wird der Speicher gesetzt oder zurückgesetzt.
5. `E_SR.Q` ⟶ `AX_SPLIT_2.IN`: der Speicherzustand wird verteilt.
6. `AX_SPLIT_2.OUT1` ⟶ `DigitalOutput_Q1.OUT`: die Lampe `Q1` folgt dem Speicher.
7. `AX_SPLIT_2.OUT2` ⟶ `E_SWITCH.G`: Rückkopplung. Ist der Speicher gesetzt, schaltet der nächste Takt ihn zurück, ist er zurückgesetzt, setzt der nächste Takt ihn. So toggelt die Lampe im Sekundentakt.
8. `STOP.IND` ⟶ `E_SR.R`: **zusätzlich** zum Stoppen des Taktgebers wird der Speicher direkt zurückgesetzt.

Verhalten: Ein Klick auf `I1` startet das Blinken, ein Klick auf `I2` stoppt es. Weil `STOP` zusätzlich direkt `E_SR.R` ansteuert, wird `Q` und damit die Lampe `Q1` beim Stoppen **zwingend** auf FALSE gesetzt - der Blinker bleibt immer auf AUS stehen.

Zusammenfassung

Die Übung löst das Problem, dass ein einfacher Blinker beim Stoppen im Zustand „EIN“ hängen bleiben könnte. Durch die Kombination aus `E_CYCLE`, der rückgekoppelten Weiche `E_SWITCH` und dem Speicher `E_SR` sowie dem zusätzlichen direkten Reset von `E_SR` durch `STOP` endet der Blinker garantiert im Zustand AUS.

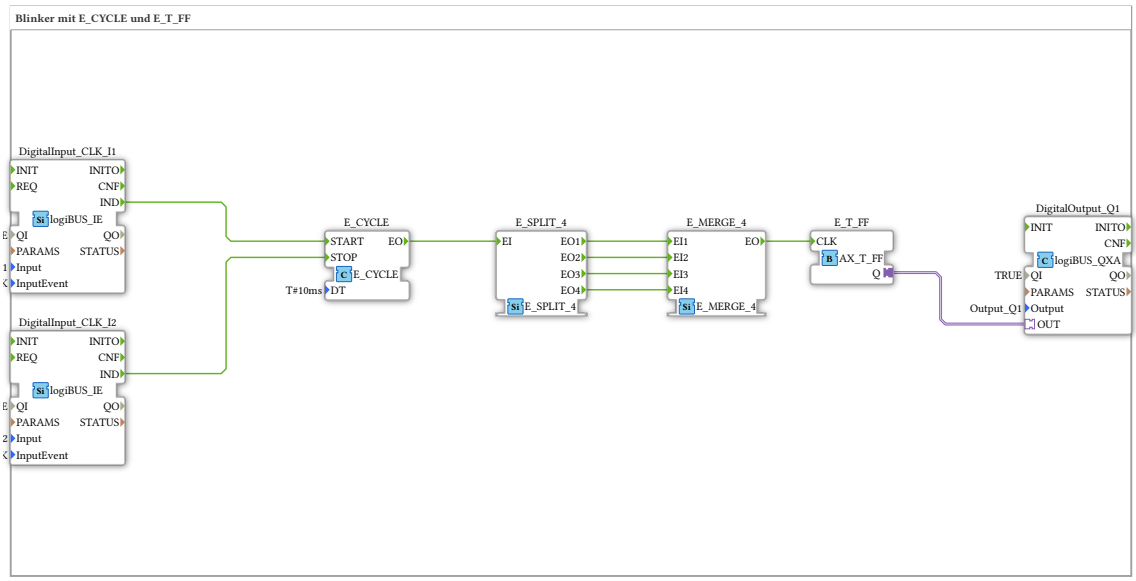
Anwendungsbeispiel - professioneller Indikator: Eine Störungsleuchte in der Industrie muss blinken, wenn ein Fehler vorliegt, und **aus** sein, wenn der Fehler quittiert wurde. Sie darf niemals dauerhaft leuchten (das würde oft „Betriebsbereit“ oder etwas anderes bedeuten).

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

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

2.1.2.107 Uebung_007b_AX: Blinker mit E_CYCLE und E_T_FF

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_007b_AX (Blinker mit E_CYCLE und E_T_FF).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Blinker mit E_CYCLE und E_T_FF**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_007b_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
- **E_CYCLE**: Instanz des Typs iec61499::events::E_CYCLE.
  - Parameter DT = T#10ms
- **E_T_FF**: Instanz des Typs adapter::events::unidirectional::AX_T_FF.
- **E_SPLIT_4**: Instanz des Typs iec61499::events::E_SPLIT_4.
- **E_MERGE_4**: Instanz des Typs iec61499::events::E_MERGE_4.
- **DigitalInput_CLK_I2**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I2
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **DigitalInput_CLK_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
  - Parameter InputEvent = BUTTON_SINGLE_CLICK

Verbindungen und Schnittstellen

Adapterverbindungen:

- E_T_FF.Q -> DigitalOutput_Q1.OUT

Ereignisverbindungen:

- E_MERGE_4.EO -> E_T_FF.CLK
- E_SPLIT_4.EO1 -> E_MERGE_4.EI1
- E_SPLIT_4.EO2 -> E_MERGE_4.EI2
- E_SPLIT_4.EO3 -> E_MERGE_4.EI3
- E_SPLIT_4.EO4 -> E_MERGE_4.EI4
- E_CYCLE.EO -> E_SPLIT_4.EI
- DigitalInput_CLK_I1.IND -> E_CYCLE.START
- DigitalInput_CLK_I2.IND -> E_CYCLE.STOP

Programmablauf und Funktionsweise

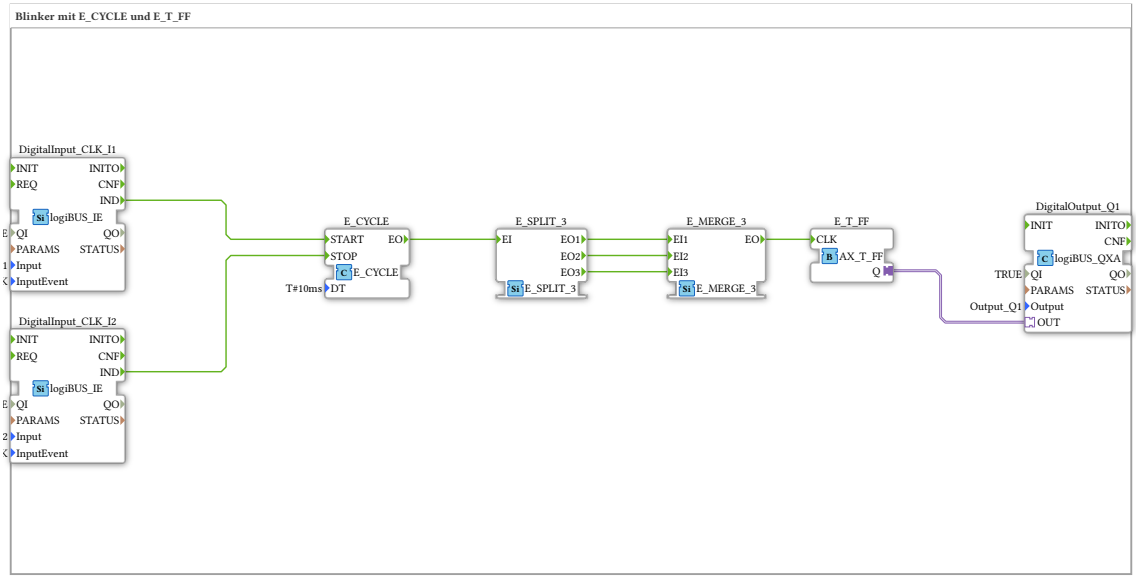
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_007b_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.108 Uebung_007c_AX: Blinker mit E_CYCLE und E_T_FF

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_007c_AX (Blinker mit E_CYCLE und E_T_FF).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Blinker mit E_CYCLE und E_T_FF**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_007c_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
- **E_CYCLE**: Instanz des Typs iec61499::events::E_CYCLE.
  - Parameter DT = T#10ms
- **E_T_FF**: Instanz des Typs adapter::events::unidirectional::AX_T_FF.
- **E_SPLIT_3**: Instanz des Typs iec61499::events::E_SPLIT_3.
- **E_MERGE_3**: Instanz des Typs iec61499::events::E_MERGE_3.
- **DigitalInput_CLK_I2**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I2
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **DigitalInput_CLK_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
  - Parameter InputEvent = BUTTON_SINGLE_CLICK

Verbindungen und Schnittstellen

Adapterverbindungen:

- E_T_FF.Q -> DigitalOutput_Q1.OUT

Ereignisverbindungen:

- E_MERGE_3.EO -> E_T_FF.CLK
- E_SPLIT_3.EO1 -> E_MERGE_3.EI1
- E_SPLIT_3.EO2 -> E_MERGE_3.EI2
- E_SPLIT_3.EO3 -> E_MERGE_3.EI3
- E_CYCLE.EO -> E_SPLIT_3.EI
- DigitalInput_CLK_I1.IND -> E_CYCLE.START
- DigitalInput_CLK_I2.IND -> E_CYCLE.STOP

Programmablauf und Funktionsweise

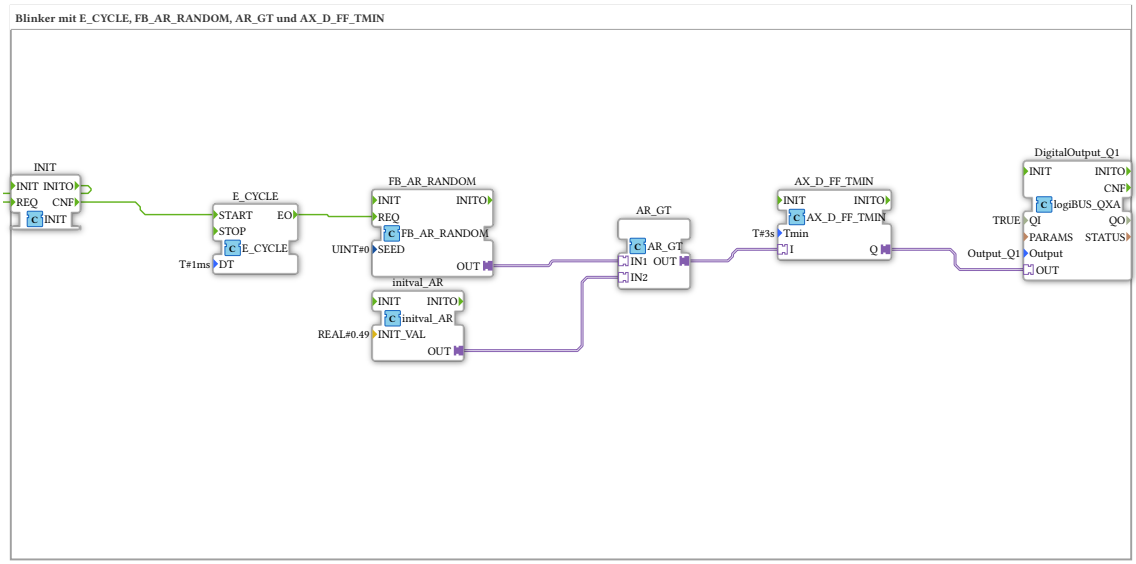
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_007c_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.109 Uebung_007d2_AX: Blinker mit E_CYCLE, FB_AR_RANDOM, AR_GT und AX_D_FF_TMIN

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_007d2_AX (Blinker mit E_CYCLE, FB_AR_RANDOM, AR_GT und AX_D_FF_TMIN).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Blinker mit E_CYCLE, FB_AR_RANDOM, AR_GT und AX_D_FF_TMIN**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_007d2_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
- **E_CYCLE**: Instanz des Typs iec61499::events::E_CYCLE.
  - Parameter DT = T#1ms
- **FB_AR_RANDOM**: Instanz des Typs adapter::utils::FB_AR_RANDOM.
  - Parameter SEED = 0
- **initval_AR**: Instanz des Typs adapter::types::unidirectional::AR::initval::initval_AR.
  - Parameter INIT_VAL = REAL#0.49
- **AR_GT**: Instanz des Typs adapter::iec61131::comparison::AR_GT.
- **AX_D_FF_TMIN**: Instanz des Typs adapter::events::unidirectional::AX_D_FF_TMIN.
  - Parameter Tmin = T#3s
- **INIT**: Instanz des Typs iec61131::booleanOperators::INIT.

Verbindungen und Schnittstellen

Adapterverbindungen:

- FB_AR_RANDOM.OUT -> AR_GT.IN1
- initval_AR.OUT -> AR_GT.IN2
- AR_GT.OUT -> AX_D_FF_TMIN.I
- AX_D_FF_TMIN.Q -> DigitalOutput_Q1.OUT

Ereignisverbindungen:

- INIT.INITO -> INIT.REQ
- INIT.CNF -> E_CYCLE.START
- E_CYCLE.EO -> FB_AR_RANDOM.REQ

Programmablauf und Funktionsweise

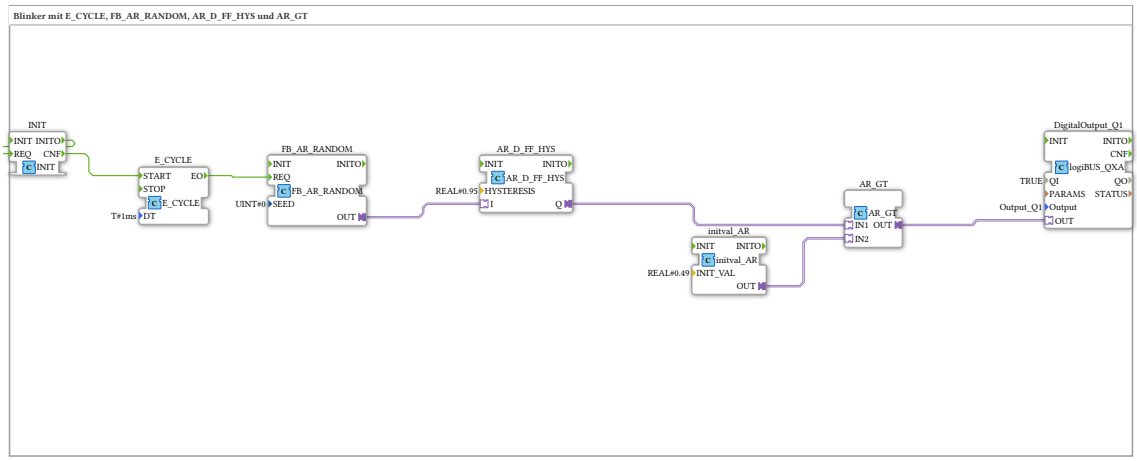
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_007d2_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.110 Uebung_007d3_AX: Blinker mit E_CYCLE, FB_AR_RANDOM, AR_D_FF_HYS und AR_GT

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_007d3_AX (Blinker mit E_CYCLE, FB_AR_RANDOM, AR_D_FF_HYS und AR_GT).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Blinker mit E_CYCLE, FB_AR_RANDOM, AR_D_FF_HYS und AR_GT**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_007d3_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1

- **E_CYCLE**: Instanz des Typs iec61499::events::E_CYCLE.
  - Parameter DT = T#1ms
- **FB_AR_RANDOM**: Instanz des Typs adapter::utils::FB_AR_RANDOM.
  - Parameter SEED = 0
- **AR_D_FF_HYS**: Instanz des Typs adapter::events::unidirectional::AR_D_FF_HYS.
  - Parameter HYSTERESIS = REAL#0.95
- **initval_AR**: Instanz des Typs adapter::types::unidirectional::AR::initval::initval_AR.
  - Parameter INIT_VAL = REAL#0.49
- **AR_GT**: Instanz des Typs adapter::iec61131::comparison::AR_GT.
- **INIT**: Instanz des Typs iec61131::booleanOperators::INIT.

Verbindungen und Schnittstellen

Adapterverbindungen:

- FB_AR_RANDOM.OUT -> AR_D_FF_HYS.I
- AR_D_FF_HYS.Q -> AR_GT.IN1
- initval_AR.OUT -> AR_GT.IN2
- AR_GT.OUT -> DigitalOutput_Q1.OUT

Ereignisverbindungen:

- INIT.INITO -> INIT.REQ
- INIT.CNF -> E_CYCLE.START
- E_CYCLE.EO -> FB_AR_RANDOM.REQ

Programmablauf und Funktionsweise

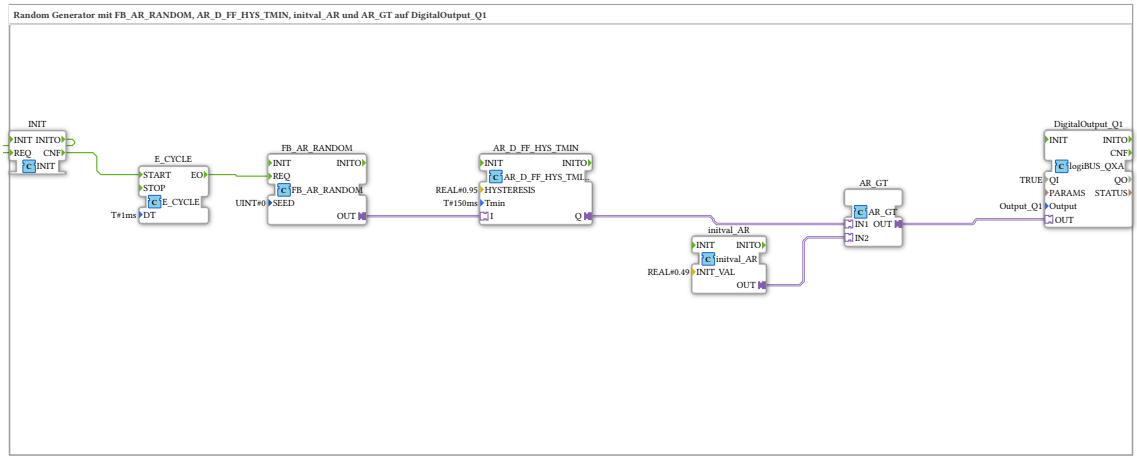
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_007d3_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.111 Uebung_007d4_AX: Random Generator mit FB_AR_RANDOM, AR_D_FF_HYS_TMIN, initval_AR und AR_GT auf DigitalOutput_Q1

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_007d4_AX (Random Generator mit FB_AR_RANDOM, AR_D_FF_HYS_TMIN, initval_AR und AR_GT auf DigitalOutput_Q1).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Random Generator mit FB_AR_RANDOM, AR_D_FF_HYS_TMIN, initval_AR und AR_GT auf DigitalOutput_Q1**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_007d4_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
- **E_CYCLE**: Instanz des Typs iec61499::events::E_CYCLE.
  - Parameter DT = T#1ms
- **FB_AR_RANDOM**: Instanz des Typs adapter::utils::FB_AR_RANDOM.
  - Parameter SEED = 0
- **AR_D_FF_HYS_TMIN**: Instanz des Typs adapter::events::unidirectional::AR_D_FF_HYS_TMIN.
  - Parameter HYSTERESIS = REAL#0.95
  - Parameter Tmin = T#150ms
- **initval_AR**: Instanz des Typs adapter::types::unidirectional::AR::initval::initval_AR.
  - Parameter INIT_VAL = REAL#0.49
- **AR_GT**: Instanz des Typs adapter::iec61131::comparison::AR_GT.
- **INIT**: Instanz des Typs iec61131::booleanOperators::INIT.

Verbindungen und Schnittstellen

Adapterverbindungen:

- FB_AR_RANDOM.OUT -> AR_D_FF_HYS_TMIN.I
- AR_D_FF_HYS_TMIN.Q -> AR_GT.IN1
- initval_AR.OUT -> AR_GT.IN2
- AR_GT.OUT -> DigitalOutput_Q1.OUT

Ereignisverbindungen:

- INIT.INITO -> INIT.REQ
- INIT.CNF -> E_CYCLE.START
- E_CYCLE.EO -> FB_AR_RANDOM.REQ

Programmablauf und Funktionsweise

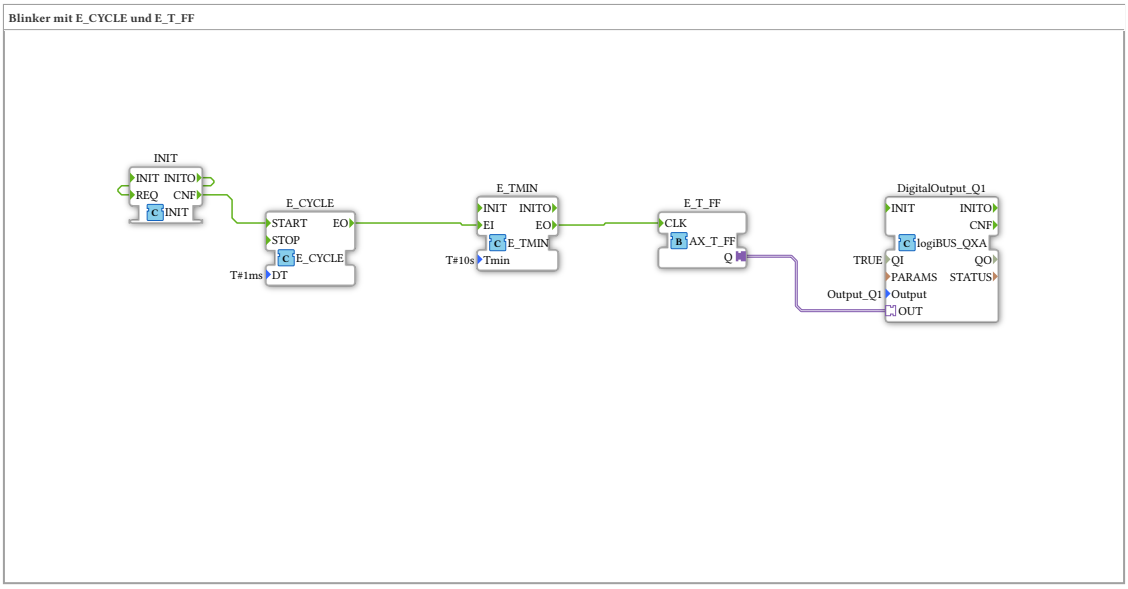
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_007d4_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.112 Uebung_007d_AX: Blinker mit E_CYCLE und E_T_FF

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_007d_AX (Blinker mit E_CYCLE und E_T_FF).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Blinker mit E_CYCLE und E_T_FF**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_007d_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
- **E_CYCLE**: Instanz des Typs iec61499::events::E_CYCLE.
  - Parameter DT = T#1ms
- **E_T_FF**: Instanz des Typs adapter::events::unidirectional::AX_T_FF.
- **E_TMIN**: Instanz des Typs iec61499::events::E_TMIN.
  - Parameter Tmin = T#10s
- **INIT**: Instanz des Typs iec61131::booleanOperators::INIT.

Verbindungen und Schnittstellen

Adapterverbindungen:

- E_T_FF.Q -> DigitalOutput_Q1.OUT

Ereignisverbindungen:

- E_CYCLE.EO -> E_TMIN.EI
- E_TMIN.EO -> E_T_FF.CLK
- INIT.INITO -> INIT.REQ
- INIT.CNF -> E_CYCLE.START

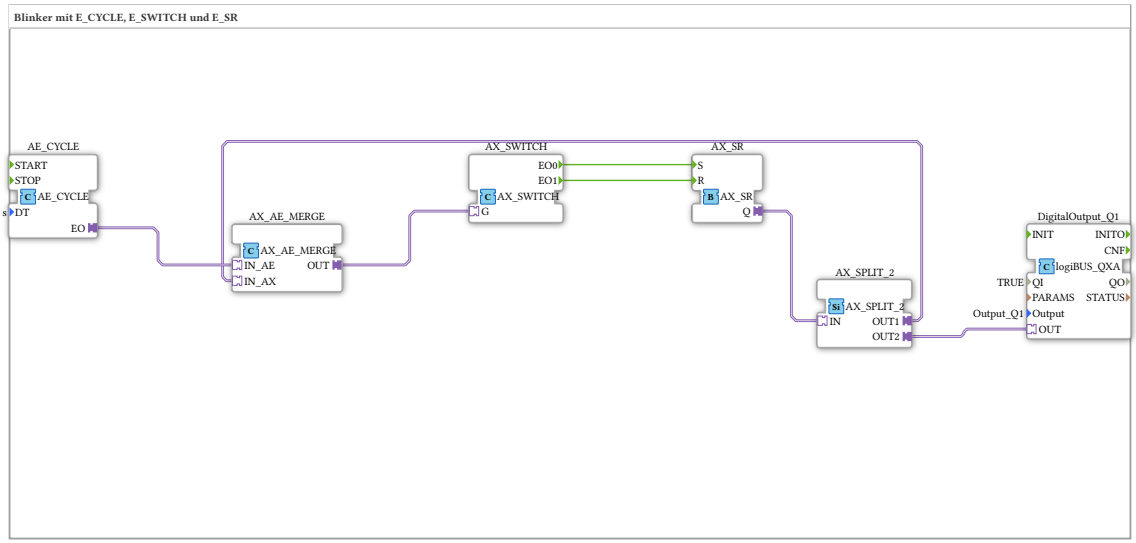
Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_007d_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.113 Uebung_008_AE: Blinker mit E_CYCLE, E_SWITCH und E_SR



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung realisiert einen einfachen Blinker (Wechselblinker) unter Verwendung von Adapter-Funktionsbausteinen (FBs) für die ereignisgesteuerte Logik. Der Kern besteht aus einem Zyklusgeber (AE_CYCLE), einem Umschalter (AX_SWITCH) und einem SR-Flipflop (AX_SR). Über einen Aufteiler (AX_SPLIT_2) und einen Zusammenführer (AX_AE_MERGE) wird die Rückkopplung und die Ausgabe auf einen digitalen Ausgang realisiert. Die Übung demonstriert die Nutzung von Ereignis- und Adapterverbindungen in der 4diac-IDE.

Verwendete Funktionsbausteine (FBs)

Alle verwendeten Bausteine sind Adapter-FBs aus den Bibliotheken `adapter::events::unidirectional::timers` und `adapter::events::unidirectional`. Zusätzlich kommt ein Hardwareausgang (logiBUS) zum Einsatz.

AE_CYCLE

- **Typ**: `adapter::events::unidirectional::timers::AE_CYCLE`
- **Parameter**:
  - `DT` = `T#1s` (Zykluszeit 1 Sekunde)
- **Ereignisausgang**:
  - `EO`: Zyklischer Ereignisimpuls alle 1 s
- **Funktion**: Erzeugt einen periodischen Ereignisimpuls. Wird als Taktgeber für den Blinker verwendet.

AX_SWITCH

- **Typ**: `adapter::events::unidirectional::AX_SWITCH`
- **Parameter**: Keine
- **Ereigniseingang**:
  - `G`: Schalteingang (Gate)
- **Ereignisausgänge**:
  - `EO0`: Aktiv, wenn das eingehende Ereignis den Zustand „Aus“ darstellt
  - `EO1`: Aktiv, wenn das eingehende Ereignis den Zustand „Ein“ darstellt
- **Funktion**: Schaltet ein eingehendes Ereignis entweder auf Ausgang 0 oder 1, abhängig vom aktuellen Zustand eines internen Umschalters. Hier wird das zurückgeführte Ereignis über `AX_AE_MERGE.OUT` an den Gate-Eingang gelegt und abwechselnd das Setzen (S) und Rücksetzen (R) des SR-Flipflops getriggert.

AX_SR

- **Typ**: `adapter::events::unidirectional::AX_SR`
- **Parameter**: Keine
- **Ereigniseingänge**:
  - `S`: Setzen (Ereignis setzt Ausgang Q auf true)
  - `R`: Rücksetzen (Ereignis setzt Ausgang Q auf false)
- **Adapterausgang**:
  - `Q`: Adapterausgang (trägt den booleschen Zustand; wird an den Aufteiler weitergegeben)
- **Funktion**: Realisiert ein SR-Flipflop auf Ereignisebene. Der Ausgang Q wird bei einem Impuls auf S gesetzt und bei einem Impuls auf R zurückgesetzt.

AX_SPLIT_2

- **Typ**: `adapter::events::unidirectional::AX_SPLIT_2`
- **Parameter**: Keine
- **Adaptereingang**:
  - `IN` (Eingang eines Adapters, der den Zustand des SR-Flipflops übernimmt)
- **Adapterausgänge**:
  - `OUT1`: Erste Kopie des Eingangs (wird zur Rückkopplung an den Schalter verwendet)
  - `OUT2`: Zweite Kopie des Eingangs (wird an den digitalen Ausgang geführt)
- **Funktion**: Verteilt den eingehenden Adapter (Zustand) auf zwei parallele Pfade.

AX_AE_MERGE

- **Typ**: `adapter::events::unidirectional::AX_AE_MERGE`
- **Parameter**: Keine
- **Adaptereingänge**:
  - `IN_AX`: Adaptereingang (Zustand vom Aufteiler)
  - `IN_AE`: Ereigniseingang (vom Zyklusgeber)
- **Adapter-/Ereignisausgang**:
  - `OUT`: Adapterausgang (kombiniert den Adapterzustand mit dem Ereignis)
- **Funktion**: Verbindet einen Adapter (Datenzustand) mit einem Ereignis. Der ausgehende Adapter enthält den Zustand von `IN_AX` und wird durch das Ereignis `IN_AE` getriggert. Dadurch wird der aktuelle Zustand des SR-Flipflops an den Schalter weitergegeben, sobald das Zyklusereignis eintrifft.

DigitalOutput_Q1

- **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
- **Parameter**:
  - `QI` = `TRUE` (immer aktiviert)
  - `Output` = `Output_Q1` (physischer Ausgangskanal)
- **Adaptereingang**:
  - `OUT` (erhält den Bool-Zustand von `AX_SPLIT_2.OUT2`)
- **Funktion**: Schaltet den physischen digitalen Ausgang Q1 entsprechend des anliegenden Bool-Werts (TRUE = ein, FALSE = aus). Der Ausgang leuchtet bei gesetztem SR-Flipflop.

Programmablauf und Verbindungen

Der Blinker arbeitet ereignisgesteuert nach folgendem Schema:

1. **Taktgeber**: Der `AE_CYCLE` sendet alle 1 Sekunde ein Ereignis auf seinem Ausgang `E0`.
2. **Ereignis-Rückführung**: Das Ereignis von `AE_CYCLE` wird mit dem aktuellen Zustand des SR-Flipflops kombiniert. Dazu wird der Zustand vom `AX_SR` über `AX_SPLIT_2` an den `AX_AE_MERGE` (Eingang `IN_AX`) geleitet. Das Ereignis von `AE_CYCLE` wird an `IN_AE` angelegt. Der `AX_AE_MERGE` erzeugt an seinem Ausgang `OUT` einen Adapter, der den Zustand trägt und zeitgleich mit dem Ereignis ausgeliefert wird.
3. **Schalter**: Der Ausgang `AX_AE_MERGE.OUT` wird an den Eingang `G` des `AX_SWITCH` gelegt. Der Schalter wertet den ankommenden Adapterzustand aus:
   - Bei Zustand `false` (Flipflop zurückgesetzt) wird ein Ereignis an `E00` ausgegeben.
   - Bei Zustand `true` (Flipflop gesetzt) wird ein Ereignis an `E01` ausgegeben.
4. **SR-Flipflop**:
   - Ein Ereignis von `AX_SWITCH.E00` gelangt an den Setzeingang `S` des `AX_SR`. Damit wird der Flipflop gesetzt → Ausgang `Q` wird `true`.

- Ein Ereignis von `AX_SWITCH.EO1` gelangt an den Rücksetzeingang `R` des `AX_SR`. Damit wird der Flipflop zurückgesetzt → Ausgang `Q` wird `false`.

5. **Ausgabe**: Über `AX_SPLIT_2` wird der Zustand `Q` auf zwei Pfade verteilt:
   - `OUT1` → Rückkopplung an `AX_AE_MERGE` (wie beschrieben)
   - `OUT2` → An den Adaptereingang `OUT` des `DigitalOutput_Q1`. Bei `true` wird der Ausgang eingeschaltet, bei `false` ausgeschaltet.

Der Kreislauf wiederholt sich mit jedem Takt von `AE_CYCLE`. Dadurch wechselt der Ausgang Q1 periodisch zwischen Ein- und Ausgeschaltet (Blinken mit 1 s Takt).

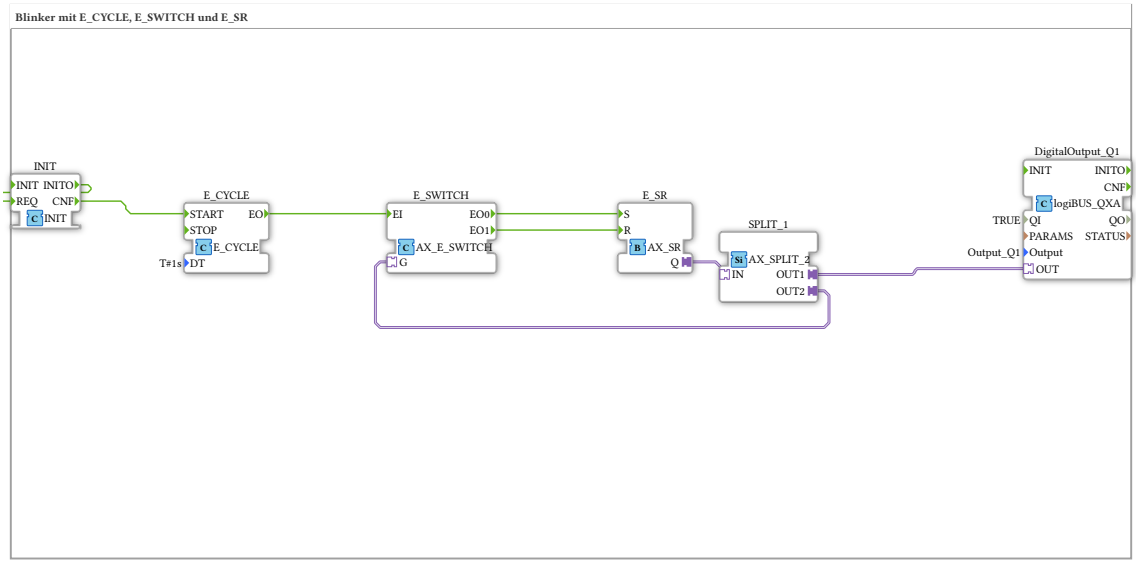
Zusammenfassung

Die Übung „Uebung_008_AE“ zeigt den Aufbau eines Blinkers mit Hilfe von 4diac-Adaptern. Zentral sind die Bausteine `AE_CYCLE` (Takt), `AX_SWITCH` (Zustandsverarbeitung), `AX_SR` (Flipflop), `AX_SPLIT_2` (Signalverteilung) und `AX_AE_MERGE` (Kombination von Ereignis und Adapter). Die ereignisbasierte Rückkopplung erzeugt eine wechselnde Ein-/Aus-Logik, die auf einen physischen Ausgang gegeben wird. Die Übung vermittelt den Umgang mit Adapterverbindungen, Ereignisrückkopplungen und dem Entwurf sequenzieller Schaltungen in der 4diac-IDE.

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

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

2.1.2.114 Uebung_008_AX: Blinker mit E_CYCLE, E_SWITCH und E_SR



• – ○ • – ○ • – ○ •



Einleitung

Implementierung eines dauerhaft laufenden Blinkers als Basis-Baustein. Ein zyklischer Takt (`E_CYCLE`, 1 s) schaltet über eine Weiche (`E_SWITCH`) abwechselnd das Setzen und das Rücksetzen eines Set/Reset-Speichers (`E_SR`) an. Der Speicherzustand wird auf den Ausgang `Q1` gelegt und zugleich als Schaltsignal auf die Weiche zurückgeführt. Da keine externen Eingriffe vorgesehen sind, blinkt der Ausgang permanent, solange die Steuerung läuft.

Verwendete Funktionsbausteine (FBs)

- **INIT**: Startauslöser
  - **Typ**: `iec61131::booleanOperators::INIT`
  - **Parameter**: keine

  - **Erklärung**: Durchleitung von Ereignissen für die Initialisierungsphase. Der Ausgang `INITO` ist auf den eigenen Eingang `REQ` zurückgeführt, die anschließende Bestätigung `CNF` startet den Takt.
- **E_CYCLE**: Taktgeber
  - **Typ**: `iec61499::events::E_CYCLE`
  - **Parameter**: `DT = T#1s`
  - **Erklärung**: Erzeugt nach dem Start im Abstand von einer Sekunde ein Ereignis `EO`.
- **E_SWITCH**: Weiche
  - **Typ**: `adapter::events::unidirectional::AX_E_SWITCH`
  - **Parameter**: keine
  - **Erklärung**: Leitet das Ereignis `EI` abhängig vom Schaltsignal am Adapter `G` an `EO0` (`G` = 0) oder `EO1` (`G` = 1) weiter.
- **E_SR**: Set/Reset-Speicher
  - **Typ**: `adapter::events::unidirectional::AX_SR`
  - **Parameter**: keine
  - **Erklärung**: Das Ereignis `S` setzt den Ausgang `Q` auf TRUE, `R` setzt ihn auf FALSE.
- **SPLIT_1**: Signalverteiler
  - **Typ**: `adapter::events::unidirectional::AX_SPLIT_2`
  - **Parameter**: keine
  - **Erklärung**: Verteilt das Signal vom Speicher `E_SR.Q` auf zwei Ausgänge: `OUT1` für den Ausgang `Q1`, `OUT2` als Rückführung zur Weiche.
- **DigitalOutput_Q1**: logiBUS Digitalausgang
  - **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
  - **Parameter**: `QI = TRUE`, `Output = Output_Q1`
  - **Erklärung**: Gibt den Zustand des Speichers an den physischen Ausgang `Q1` aus.

Programmablauf und Verbindungen

1. `INIT.INITO` ⟶ `INIT.REQ` (EventConnection): Die Initialisierungsbestätigung löst die Dienstanforderung am selben Baustein aus.
2. `INIT.CNF` ⟶ `E_CYCLE.START` (EventConnection): Der Takt wird gestartet.
3. `E_CYCLE.EO` ⟶ `E_SWITCH.EI` (EventConnection): Jeder Taktimpuls (alle 1 s) geht in die Weiche.
4. `E_SWITCH.EO0` ⟶ `E_SR.S` und `E_SWITCH.EO1` ⟶ `E_SR.R` (EventConnections): Bei `G = 0` wird der Speicher gesetzt, bei `G = 1` zurückgesetzt.
5. `E_SR.Q` ⟶ `SPLIT_1.IN` (AdapterConnection): Der Speicherzustand geht in den Verteiler.
6. `SPLIT_1.OUT1` ⟶ `DigitalOutput_Q1.OUT` (AdapterConnection): Der Zustand wird auf den physischen Ausgang `Q1` gelegt.
7. `SPLIT_1.OUT2` ⟶ `E_SWITCH.G` (AdapterConnection): Der Zustand wird als Schaltsignal auf die Weiche zurückgeführt.

Verhalten: Ist der Speicher zurückgesetzt (`Q` = FALSE, `G` = 0), setzt der nächste Taktimpuls ihn über `EO0`/`S`; ist er gesetzt (`G` = 1), setzt der nächste Impuls ihn über `EO1`/`R` zurück. Der Ausgang `Q1` wechselt dadurch jede Sekunde seinen Zustand.

Anwendungsbeispiel

Lebenszeichen (Heartbeat): Eine LED, die auf der Platine oder am Schaltschrank blinkt, um anzuzeigen: „Die CPU lebt noch und das Programm läuft“.

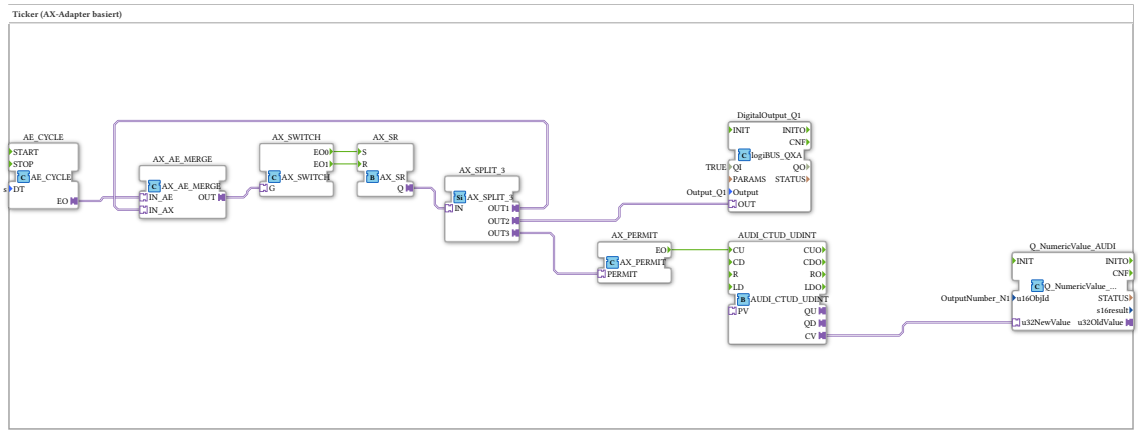
Zusammenfassung

Die Übung zeigt einen Blinker aus Standardbausteinen: `E_CYCLE` liefert den Takt, `E_SWITCH` und `E_SR` bilden mit der Rückführung über `SPLIT_1` eine Toggle-Logik, und `INIT` startet den Takt nach der Initialisierung.

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

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

2.1.2.115 Uebung_009_AUDI: Ticker (AX-Adapter basiert)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert die Realisierung eines **Tickers** auf Basis von **AX-Adaptern** (Adapter-Ereignis-Schnittstellen) in der 4diac-IDE. Ziel ist es, einen Zähler (up/down) zu implementieren, der über einen Schalter (AX_SWITCH), ein Set-Reset-Glied (AX_SR) und ein Erlaubnissignal (AX_PERMIT) gesteuert wird. Der aktuelle Zählerstand wird auf einem digitalen Ausgang und einem numerischen Anzeigeblock ausgegeben.

Die Übung ist Teil der **Uebungen**-Bibliothek und nutzt vordefinierte Adapterbausteine sowie einen CTUD-Zähler. Sie eignet sich für Fortgeschrittene, die das Zusammenspiel von Ereignissen und Adaptern verstehen möchten.

Verwendete Funktionsbausteine (FBs)

Die Übung besteht aus einer **SubApp** (Uebung_009_AUDI), die mehrere interne Funktionsbausteine enthält. Die SubApp selbst hat keine eigenen Ein-/Ausgangsschnittstellen; alle Signale werden über interne Verbindungen verarbeitet.

Sub-Bausteine: Uebung_009_AUDI (SubApp)

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **DigitalOutput_Q1**: `logiBUS::io::DQ::logiBUS_QXA`
    - Parameter: `QI` = `TRUE`, `Output` = `Output_Q1`
    - Funktion: Stellt den digitalen Ausgang `Output_Q1` auf dem logiBUS bereit.
  - **AE_CYCLE**: `adapter::events::unidirectional::timers::AE_CYCLE`
    - Parameter: `DT` = `T#1s`
    - Funktion: Erzeugt zyklisch alle 1 Sekunde ein Ereignis an seinem Ausgang `EO`.
  - **AX_SWITCH**: `adapter::events::unidirectional::AX_SWITCH`
    - Parameter: keine
    - Funktion: Ein AX-Adapter-Schalter, der zwei Ereignisausgänge (`EO0`, `EO1`) besitzt. Welcher Ausgang aktiv wird, hängt vom eingehenden Adapter-Ereignis ab (Toggle-Funktion).
  - **AX_SR**: `adapter::events::unidirectional::AX_SR`
    - Parameter: keine
    - Funktion: Set-Reset-Speicher mit AX-Adapter-Schnittstelle. Die Eingänge `S` und `R` setzen bzw. rücksetzen den Ausgang `Q`.
  - **AX_PERMIT**: `adapter::events::unidirectional::AX_PERMIT`
    - Parameter: keine
    - Funktion: Ein Erlaubnis-Glied: Nur wenn am Eingang `PERMIT` ein Ereignis eintrifft, wird das an `IN` anliegende Ereignis an den Ausgang `EO` weitergeleitet.
  - **AUDI_CTUD_UDINT**: `adapter::events::unidirectional::AUDI_CTUD_UDINT`
    - Parameter: keine
    - Funktion: Zähler mit Vorwärtszählen (CU) und optionale Zählrichtung. Liefert den aktuellen Zählerwert als `UDINT` an `CV`.
  - **Q_NumericValue_AUDI**: `isobus::UT::Q::Q_NumericValue_AUDI`

    - Parameter: `u16ObjId` = `OutputNumber_N1`
    - Funktion: Gibt den übergebenen Zahlenwert (`u32NewValue`) in ein isobus-Netzwerk aus (Objekt-ID `OutputNumber_N1`).
- **AX_SPLIT_3**: `adapter::events::unidirectional::AX_SPLIT_3`
    - Parameter: keine
    - Funktion: Verteilt ein eingehendes AX-Ereignis an drei Ausgänge (`OUT1`, `OUT2`, `OUT3`).
- **AX_AE_MERGE**: `adapter::events::unidirectional::AX_AE_MERGE`
    - Parameter: keine
    - Funktion: Vereinigt zwei Ereigniseingänge: einen AX-Adapter (`IN_AX`) und einen reinen Ereigniseingang (`IN_AE`). Das kombinierte Signal wird am Ausgang `OUT` ausgegeben.

Programmablauf und Verbindungen

Der Ablauf der Übung lässt sich wie folgt beschreiben:

1. **Taktgenerierung** `AE_CYCLE` erzeugt alle 1 Sekunde ein Ereignis (EO).
2. **Ereignisvereinigung** Dieses Ereignis wird zusammen mit dem Signal von `AX_SPLIT_3.OUT1` (siehe Schritt 4) über `AX_AE_MERGE` zusammengeführt. Das Ergebnis wird an `AX_SWITCH.G` (Gate-Eingang) weitergeleitet.
3. **Schalterbetrieb** `AX_SWITCH` reagiert auf das eingehende Ereignis und schaltet zwischen seinen beiden Ausgängen `EO0` und `EO1` um. Dies simuliert ein manuelles oder logisches Umschalten.
4. **Set-Reset-Glied** `EO0` geht an `AX_SR.S` (Set), `EO1` an `AX_SR.R` (Reset). Der Ausgang `Q` des SR-Glieds wird aktiv, solange gesetzt, und deaktiviert bei Reset.
5. **Signalverteilung** Das Signal von `AX_SR.Q` wird auf `AX_SPLIT_3.IN` gegeben und auf drei Ausgänge verteilt:
    - `OUT1` ⟶ zurück zur Ereignisvereinigung `AX_AE_MERGE.IN_AX`.
    - `OUT2` ⟶ an den **digitalen Ausgang** `DigitalOutput_Q1.OUT`. Damit wird der Ausgang `Output_Q1` gesetzt, solange das SR-Glied gesetzt ist.
    - `OUT3` ⟶ an `AX_PERMIT.PERMIT`.
6. **Erlaubnis und Zähler** `AX_PERMIT` gibt das Ereignis nur dann an `EO` weiter, wenn am `PERMIT`-Eingang ein Ereignis anliegt. Dieses wird an den Zähler `AUDI_CTUD_UDINT.CU` gesendet. Der Zähler erhöht seinen Wert bei jedem Ereignis.
7. **Numerische Ausgabe** Der aktuelle Zählerstand (`CV`) wird an den `Q_NumericValue_AUDI`-Block übergeben und als numerischer Wert auf dem isobus-Netzwerk (Objekt-ID `OutputNumber_N1`) ausgegeben.

Lernziele:

- Verständnis von AX- und AE-Adaptern (Ereignis- und Adapter-Schnittstellen)
- Anwendung eines SR-Speichers, eines Schalters und eines Erlaubnisglieds
- Verknüpfung von zyklischen Ereignissen mit manueller Steuerung
- Ausgabe auf digitalen und numerischen Kanälen

Schwierigkeitsgrad: Fortgeschritten **Vorkenntnisse**: Grundlagen der 4diac-IDE, Ereignisgesteuerte Abläufe, Arbeit mit Adaptern

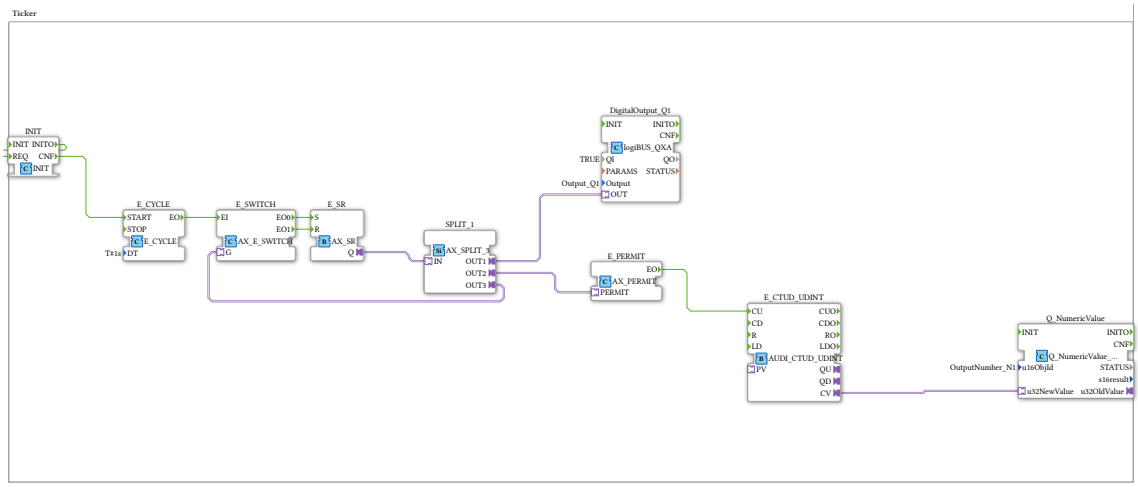
Zusammenfassung

Die Übung `Uebung_009_AUDI` implementiert einen tickergesteuerten Zähler mit Hilfe von AX-Adapter-Bausteinen. Ein zyklischer Timer (`AE_CYCLE`) liefert den Takt, der über einen Schalter (`AX_SWITCH`) und ein Set-Reset-Glied (`AX_SR`) den Zähler freigibt. Der Zählerstand wird gleichzeitig als digitales Signal auf einem logiBUS-Ausgang und als numerischer Wert auf einem isobus-Netzwerk ausgegeben. Die Verwendung von Adaptern erlaubt eine flexible, ereignisorientierte Verkettung und demonstriert die modulare Struktur der 4diac-IDE.

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

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

2.1.2.116 Uebung_009_AX: Ticker (AX-Adapter basiert)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert einen adapterbasierten Ticker: Ein Taktgeber lässt einen Ausgang im Sekundentakt blinken, jede zweite Taktflanke (Ausgang geht auf TRUE) wird von einem Auf-/Abwärtszähler gezählt, und der Zählerstand wird als Zahl auf dem ISOBUS-Terminal angezeigt. Durch die `AX`/`AUDI`-Adapter werden Ereignis- und Datenfluss jeweils in einer einzigen Verbindung gebündelt.

Verwendete Funktionsbausteine (FBs)

- **INIT**: Startauslöser
  - **Typ**: `iec61131::booleanOperators::INIT`
  - **Parameter**: keine
  - **Erklärung**: Leitet `INIT` unverzögert als `INITO` und `REQ` als `CNF` weiter. `INITO` ist auf `REQ` zurückgeführt, `CNF` startet den Taktgeber.
- **E_CYCLE**: Taktgeber
  - **Typ**: `iec61499::events::E_CYCLE`
  - **Parameter**: `DT = T#1s`
  - **Erklärung**: Erzeugt nach dem Start (`START`) jede Sekunde ein Ereignis `EO`.
- **E_SWITCH**: adapterbasierte Ereignisweiche
  - **Typ**: `adapter::events::unidirectional::AX_E_SWITCH`
  - **Parameter**: keine
  - **Erklärung**: Leitet das Takt-Ereignis `EI` je nach Wert des Adapters `G` an `EO0` (`G` = FALSE) oder `EO1` (`G` = TRUE) weiter.
- **E_SR**: adapterbasiertes Speicherglied (Blinkerstatus)
  - **Typ**: `adapter::events::unidirectional::AX_SR`
  - **Parameter**: keine
  - **Erklärung**: `S` (aus `E_SWITCH.EO0`) setzt, `R` (aus `E_SWITCH.EO1`) setzt zurück. Der Zustand wird über den Adapterausgang `Q` ausgegeben.
- **SPLIT_1**: Signalverteiler
  - **Typ**: `adapter::events::unidirectional::AX_SPLIT_3`
  - **Parameter**: keine
  - **Erklärung**: Verteilt das `AX`-Signal von `E_SR.Q` auf drei Ziele: `OUT1` (Digitalausgang), `OUT2` (Zählfreigabe) und `OUT3` (Rückführung an die Weiche).
- **E_PERMIT**: adapterbasiertes Ereignis-Gatter
  - **Typ**: `adapter::events::unidirectional::AX_PERMIT`
  - **Parameter**: keine
  - **Erklärung**: Löst `EO` nur aus, wenn am Adapter `PERMIT` ein Ereignis (`PERMIT.E1`) eintrifft und `PERMIT.D1` `TRUE` ist.
- **E_CTUD_UDINT**: adapterbasierter Vorwärts-/Rückwärtszähler
  - **Typ**: `adapter::events::unidirectional::AUDI_CTUD_UDINT`
  - **Parameter**: keine

  - **Erklärung**: Erhöht bei jedem `CU`-Ereignis den Zählwert um 1 und gibt ihn über den `AUDI`-Adapter `CV` aus.
- **Q_NumericValue**: Anzeige auf dem ISOBUS-Terminal
  - **Typ**: `isobus::UT::Q::Q_NumericValue_AUDI`
  - **Parameter**: `u16ObjId = OutputNumber_N1`
  - **Erklärung**: Empfängt den Zählwert über den `AUDI`-Adapter `u32NewValue` und zeigt ihn im Objekt `OutputNumber_N1` an.
- **DigitalOutput_Q1**: logiBUS Digitalausgang
  - **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
  - **Parameter**: `QI = TRUE`, `Output = Output_Q1`
  - **Erklärung**: Gibt den Blinkerzustand an den physischen Ausgang `Q1` weiter.

Programmablauf und Verbindungen

1. `INIT.INITO` ⟶ `INIT.REQ` und `INIT.CNF` ⟶ `E_CYCLE.START` (EventConnections): Der Taktgeber wird beim Initialisieren gestartet.
2. `E_CYCLE.EO` ⟶ `E_SWITCH.EI` (EventConnection): Jede Sekunde trifft ein Takt-Ereignis an der Weiche ein.
3. `E_SWITCH.EO0` ⟶ `E_SR.S` und `E_SWITCH.EO1` ⟶ `E_SR.R` (EventConnections): Ist das Speicherglied zurückgesetzt, setzt der Takt es, andernfalls setzt der Takt es zurück. Der Zustand wechselt also mit jedem Takt.
4. `E_SR.Q` ⟶ `SPLIT_1.IN` (AdapterConnection): Der Zustand wird auf drei Ziele verteilt.
5. `SPLIT_1.OUT3` ⟶ `E_SWITCH.G` (AdapterConnection): Der aktuelle Zustand steuert die Weiche und erzeugt damit das Umschalten.
6. `SPLIT_1.OUT1` ⟶ `DigitalOutput_Q1.OUT` (AdapterConnection): `Q1` blinkt im Rhythmus des Zustands.
7. `SPLIT_1.OUT2` ⟶ `E_PERMIT.PERMIT` (AdapterConnection): Bei jeder Zustandsänderung trifft ein Ereignis am Gatter ein; es wird nur durchgelassen, wenn der Zustand `TRUE` ist.
8. `E_PERMIT.EO` ⟶ `E_CTUD_UDINT.CU` (EventConnection): Jedes durchgelassene Ereignis zählt den Zähler um 1 hoch.
9. `E_CTUD_UDINT.CV` ⟶ `Q_NumericValue.u32NewValue` (AdapterConnection): Der Zählwert wird an das ISOBUS-Terminal gesendet.

Verhalten: `Q1` ist abwechselnd je eine Sekunde ein- und ausgeschaltet. Der Zähler wird jeweils beim Einschalten erhöht, also einmal alle zwei Sekunden; auf dem Terminal steigt die Zahl in `OutputNumber_N1` stetig an.

Zusammenfassung

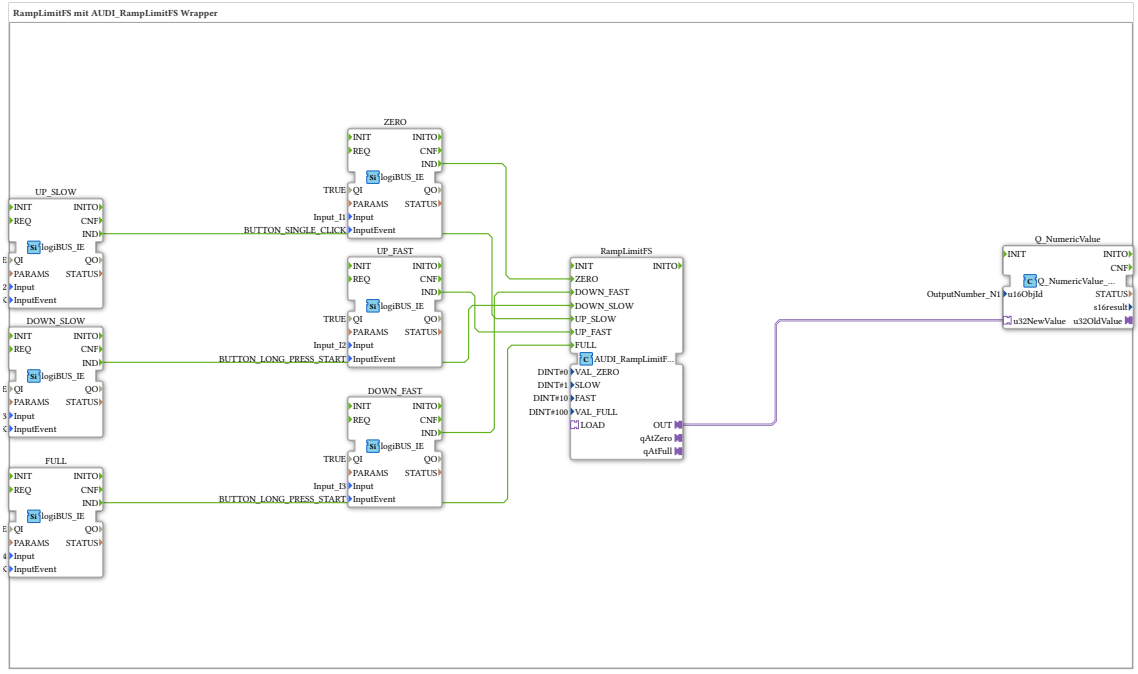
Die Übung zeigt die Vorteile einer konsequent adapterbasierten Entwicklung: Takt, Blinkerzustand, Zähler und Anzeige sind mit wenigen Adapterverbindungen verschaltet, weil Ereignis- und Datenfluss jeweils gebündelt übertragen werden. `AX_SPLIT_3` verteilt den Blinkerzustand gleichzeitig auf Ausgang, Zählfreigabe und Rückführung.

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

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

2.1.2.117 Uebung_009a_AX: RampLimitFS mit AUDI_RampLimitFS Wrapper

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_009a_AX (RampLimitFS mit AUDI_RampLimitFS Wrapper).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **RampLimitFS mit AUDI_RampLimitFS Wrapper**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_009a_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **Q_NumericValue**: Instanz des Typs isobus::UT::Q::Q_NumericValue_AUDI.
  - Parameter u16ObjId = OutputNumber_N1
- **UP_FAST**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter PARAMS =
  - Parameter Input = Input_I2
  - Parameter InputEvent = BUTTON_LONG_PRESS_START
- **DOWN_SLOW**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter PARAMS =
  - Parameter Input = Input_I3
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **ZERO**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter PARAMS =
  - Parameter Input = Input_I1
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **DOWN_FAST**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter PARAMS =
  - Parameter Input = Input_I3
  - Parameter InputEvent = BUTTON_LONG_PRESS_START
- **UP_SLOW**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter PARAMS =
  - Parameter Input = Input_I2

  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **RampLimitFS**: Instanz des Typs adapter::signalprocessing::ramp::AUDI_RampLimitFS.
  - Parameter VAL_ZERO = DINT#0
  - Parameter SLOW = DINT#1
  - Parameter FAST = DINT#10
  - Parameter VAL_FULL = DINT#100
- **FULL**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter PARAMS =
  - Parameter Input = Input_I4
  - Parameter InputEvent = BUTTON_SINGLE_CLICK

Verbindungen und Schnittstellen

Adapterverbindungen:

- RampLimitFS.OUT -> Q_NumericValue.u32NewValue

Ereignisverbindungen:

- UP_FAST.IND -> RampLimitFS.UP_FAST
- FULL.IND -> RampLimitFS.FULL
- ZERO.IND -> RampLimitFS.ZERO
- DOWN_SLOW.IND -> RampLimitFS.DOWN_SLOW
- DOWN_FAST.IND -> RampLimitFS.DOWN_FAST
- UP_SLOW.IND -> RampLimitFS.UP_SLOW

Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

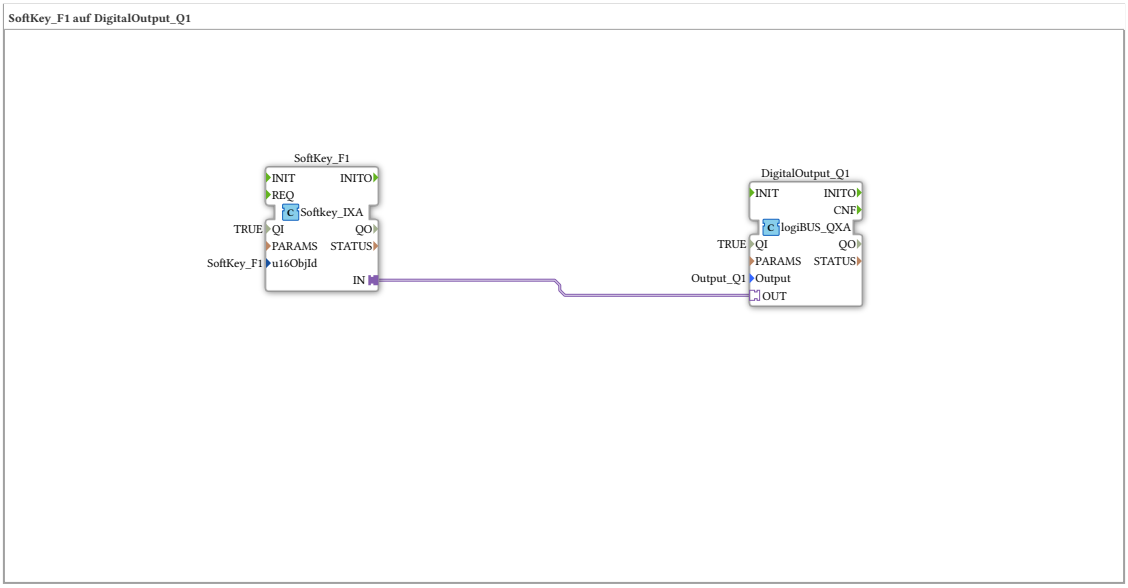
Die Übung Uebung_009a_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.118 Uebung_010_AX: SoftKey_F1 auf DigitalOutput_Q1



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_010_AX`. Hier betreten wir die Welt des ISOBUS (ISO 11783). Anstelle von physischen Eingängen nutzen wir virtuelle Tasten auf einem Terminal (Universal Terminal, UT).

Podcast



- ISO 11783-6: Softkeys und das Virtual Terminal verstehen – Dein Schlüssel zur Landmaschinen-Mechatronik

---

Ziel der Übung

Verwendung eines `Softkey`-Bausteins zur Steuerung eines Ausgangs.

Beschreibung und Komponenten

Die Subapplikation `Uebung_010_AX.SUB` verbindet eine Softkey-Instanz mit einem digitalen Ausgang.

Funktionsbausteine (FBs)

- **`SoftKey_F1`**: Typ `isobus::UT::io::Softkey::Softkey_IXA`. Dieser Baustein repräsentiert die Taste „F1“ auf dem Bildschirm des ISOBUS-Terminals.
- **`DigitalOutput_Q1`**: Der physische Ausgang.

Parameter

- `u16ObjId`: Verweist auf die Objekt-ID des Softkeys im Objekt-Pool (hier `SoftKey_F1`).

Funktionsweise

Die Funktionsweise ist identisch zu einem physischen Taster. Solange der Nutzer den Finger auf dem Touchscreen (oder die Taste am Rand) hält, liefert der Baustein `TRUE`. Lässt er los, wird `FALSE` gesendet.

Anwendungsbeispiel

Maschinenfunktion einschalten: Der Fahrer drückt auf dem Bildschirm das Symbol für „Arbeitsscheinwerfer“, und das Licht geht an (solange er drückt).

2.1.2.119 Uebung_010a2_AX: Button_A1 auf DigitalOutput_Q1



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_010a2_AX`. Neben Softkeys (F-Tasten) gibt es im ISOBUS auch „Buttons“ (Schaltflächen im Datenmasken-Bereich).

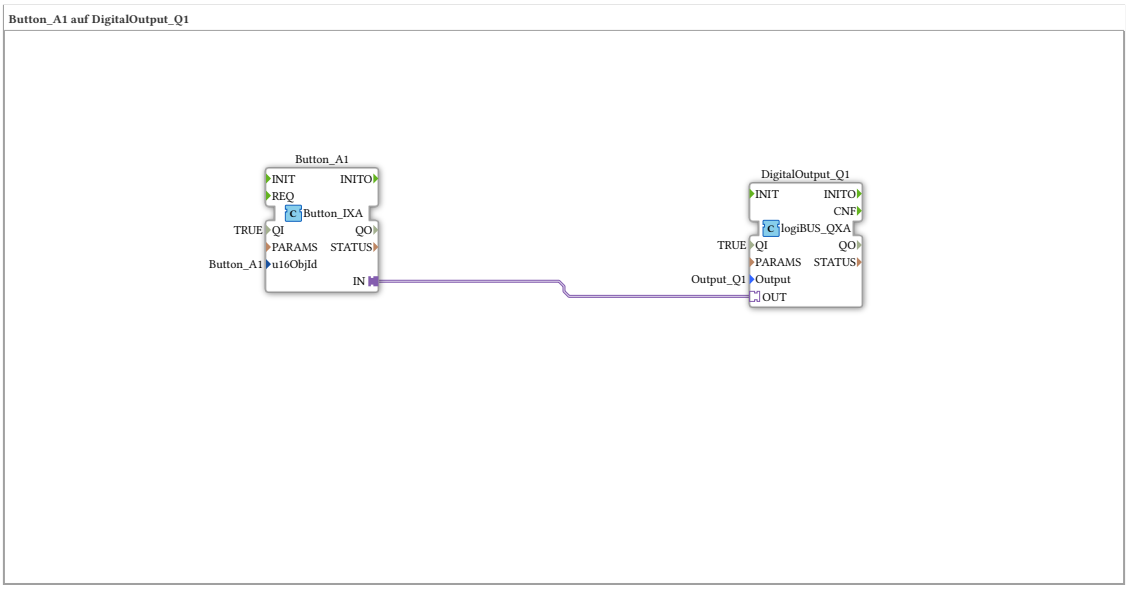
Ziel der Übung

Verwendung eines `Button_IXA`.

Beschreibung und Komponenten

Die Subapplikation `Uebung_010a2_AX.SUB` nutzt einen Button anstelle eines Softkeys.

Funktionsbausteine (FBs)



- **`Button_A1`**: Typ `isobus::UT::io::Button::Button_IXA`. Referenziert `Button_A1`.

Funktionsweise

Funktional identisch zum Softkey, jedoch visuell an einem anderen Ort auf dem Terminal. Ein „Button“ ist Teil der Arbeitsmaske, ein „Softkey“ ist Teil der fixen Menüleiste.

2.1.2.120 Uebung_010a3_AX: Button_L1 (Latching) auf DigitalOutput_Q1

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_010a3_AX`.

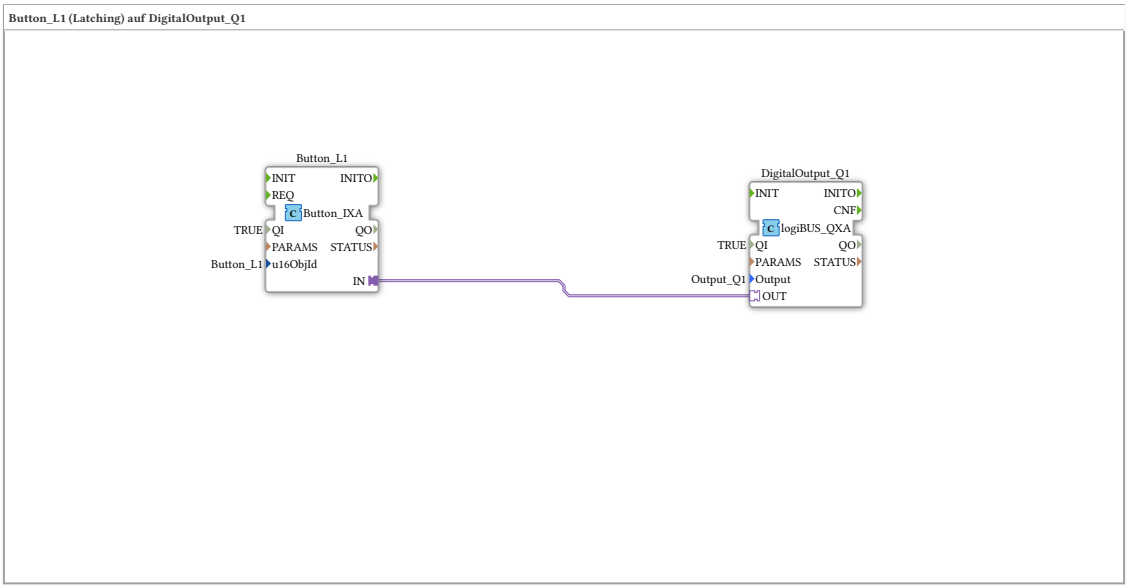
Ziel der Übung

Umgang mit rastenden Tasten (Latching Buttons).

Beschreibung und Komponenten

Die Subapplikation `Uebung_010a3_AX.SUB` verwendet `Button_L1`.

Funktionsbausteine (FBs)



- **`Button_L1`**: Ein Button, der im ISOBUS-Pool als „Latching“ definiert ist.

Funktionsweise

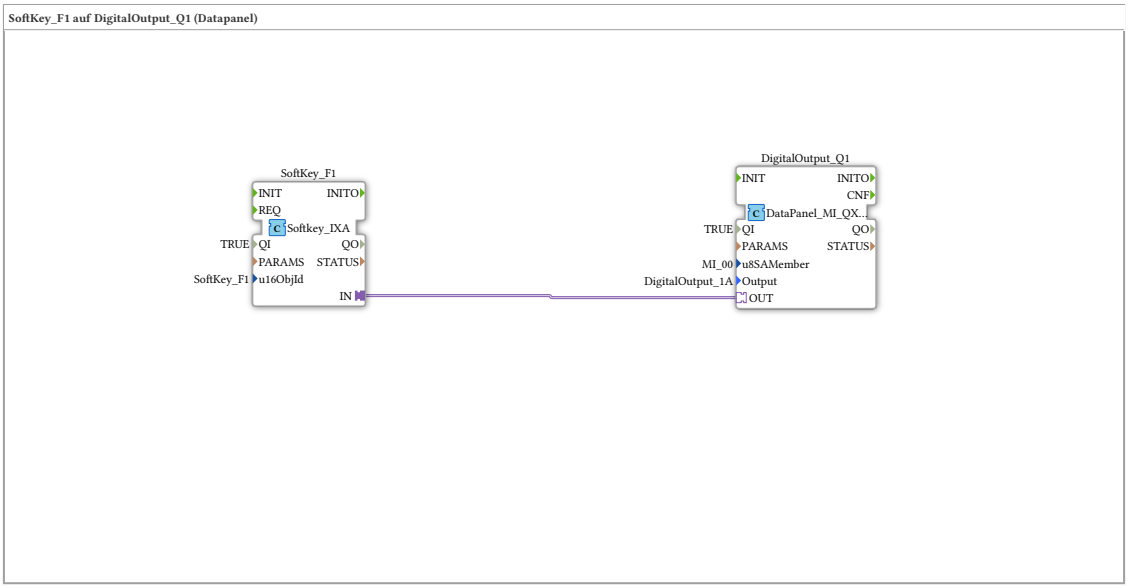
Wie im Code kommentiert: *„Latching Button ‚rastet ein‘, kein T_FF nötig!“*. Wenn der Nutzer diesen Button drückt, wechselt er seinen Zustand (Visuell z.B. eingedrückt) und sendet dauerhaft `TRUE`. Beim nächsten Druck wechselt er auf `FALSE`. Das Speicherverhalten liegt hier also im **Terminal**, nicht in der SPS-Logik.

2.1.2.121 Uebung_010a4_AX: SoftKey_F1 auf DigitalOutput_Q1 (Datapanel)



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_010a4_AX`.

🎧 Podcast



- ISO 11783-6: Softkeys und das Virtual Terminal verstehen – Dein Schlüssel zur Landmaschinen-Mechatronik

Ziel der Übung

Verknüpfung von ISOBUS (UT) und Hardware-Peripherie (DataPanel).

Beschreibung und Komponenten

Die Subapplikation `Uebung_010a4_AX.SUB` verbindet einen Softkey mit einem Ausgang eines DataPanels.

Funktionsbausteine (FBs)

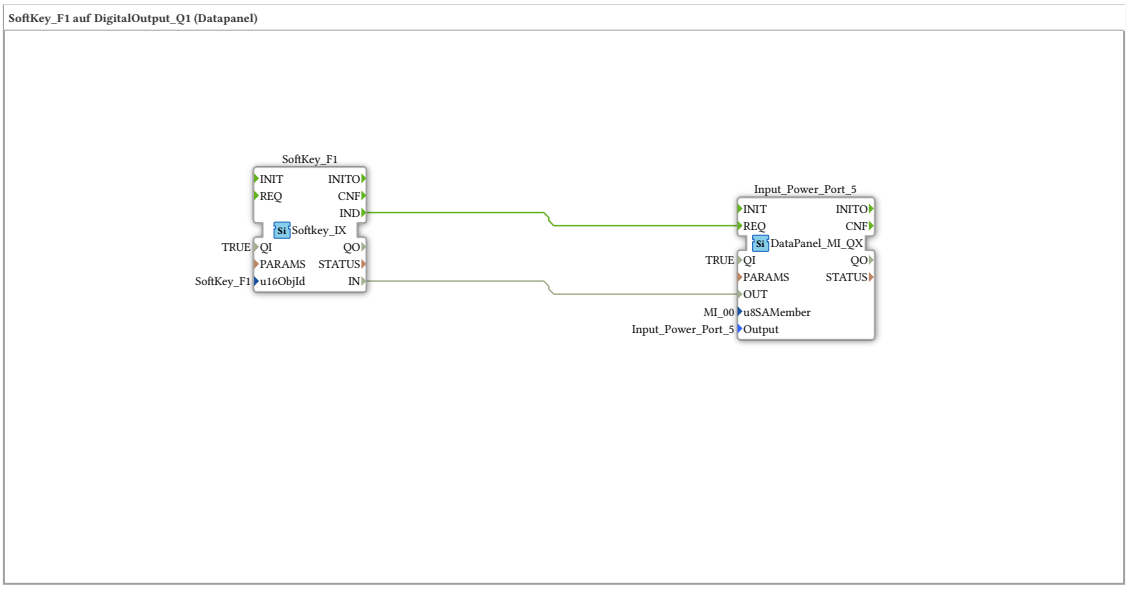
- **`SoftKey_F1`**: Eingabe via Terminal.
- **`DigitalOutput_Q1`**: Typ `DataPanel_MI_QXA`. Repräsentiert einen Ausgang auf einem externen CAN-Bus-Modul (DataPanel).

Funktionsweise

Dies demonstriert die Transparenz von logiBUS. Es ist für die Logik egal, ob der Ausgang direkt an der Steuerung sitzt (`logiBUS_QXA`) oder über CAN angebunden ist (`DataPanel_MI_QXA`). Der Adapter verbindet beides nahtlos.

2.1.2.122 Uebung_010a5_AX: SoftKey_F1 auf DigitalOutput_Q1 (Datapanel)

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_010a5_AX (SoftKey_F1 auf DigitalOutput_Q1 (Datapanel)).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **SoftKey_F1 auf DigitalOutput_Q1 (Datapanel)**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_010a5_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **Input_Power_Port_5**: Instanz des Typs DataPanel::io::MI::DQ::DataPanel_MI_QX.
  - Parameter QI = TRUE
  - Parameter u8SAMember = MI_00
  - Parameter Output = Input_Power_Port_5
- **SoftKey_F1**: Instanz des Typs isobus::UT::io::Softkey::Softkey_IX.
  - Parameter QI = TRUE

- Parameter u16ObjId = SoftKey_F1

Verbindungen und Schnittstellen
Ereignisverbindungen:

- SoftKey_F1.IND -> Input_Power_Port_5.REQ

Datenverbindungen:

- SoftKey_F1.IN -> Input_Power_Port_5.OUT

Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

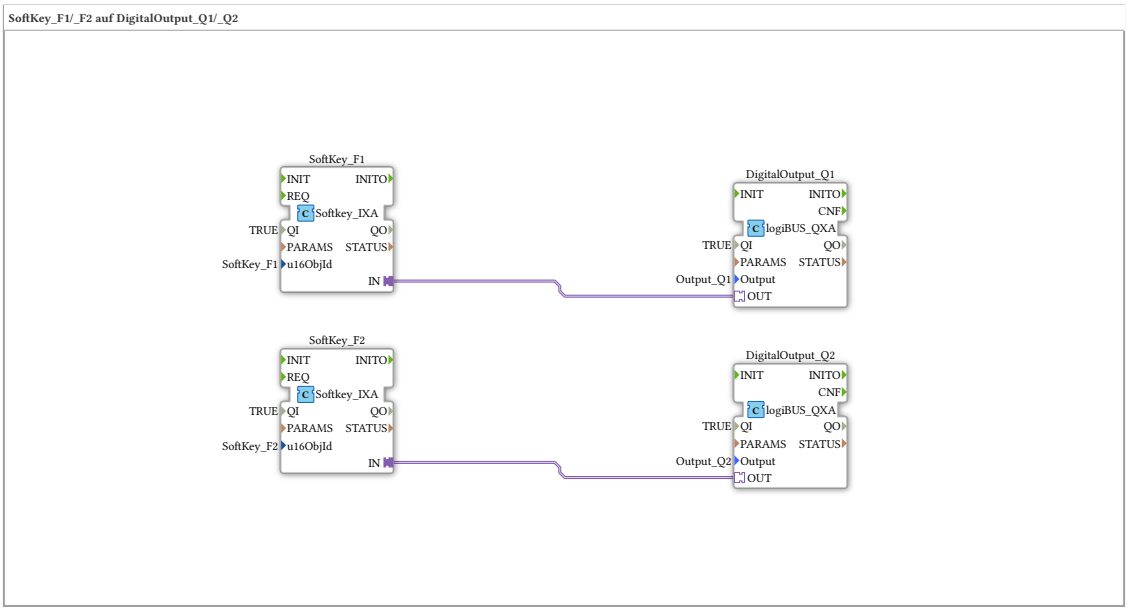
Die Übung Uebung_010a5_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.123 Uebung_010a_AX: SoftKey_F1/_F2 auf DigitalOutput_Q1/_Q2

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_010a_AX`.

Podcast



- ISO 11783-6: Softkeys und das Virtual Terminal verstehen – Dein Schlüssel zur Landmaschinen-Mechatronik

---

Ziel der Übung

Erweiterung auf mehrere Softkeys.

Beschreibung und Komponenten

Die Subapplikation `Uebung_010a_AX.SUB` steuert zwei Ausgänge über zwei Softkeys.

Funktionsbausteine (FBs)

- **`SoftKey_F1`** -> **`DigitalOutput_Q1`**
- **`SoftKey_F2`** -> **`DigitalOutput_Q2`**

Funktionsweise

Zwei unabhängige Signalpfade. Zeigt, dass man beliebig viele Softkeys instanziieren kann, solange sie im Objekt-Pool definiert sind.

2.1.2.124 Uebung_010b1_AX: AuxFunction2_X1 auf DigitalOutput_Q1



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_010b1_AX`. Neben Softkeys und Buttons ist AUX-N die dritte wichtige Eingabemethode im ISOBUS.

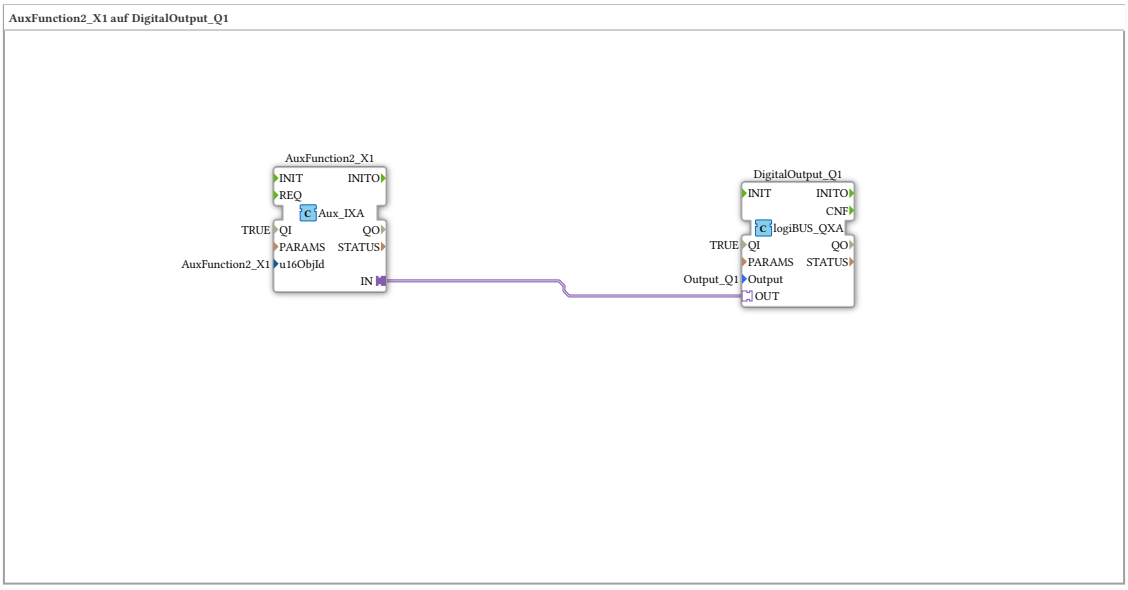
Ziel der Übung

Verarbeitung von Auxiliary Inputs (z.B. Joystick-Tasten).

Beschreibung und Komponenten

Die Subapplikation `Uebung_010b1_AX.SUB` verbindet eine AUX-Funktion mit einem Ausgang.

Funktionsbausteine (FBs)



- **`AuxFunction2_X1`**: Typ `isobus::UT::io::Auxiliary::IN::Aux_IXA`. Dieser Baustein lauscht auf ISOBUS AUX-Nachrichten für die definierte Funktion.

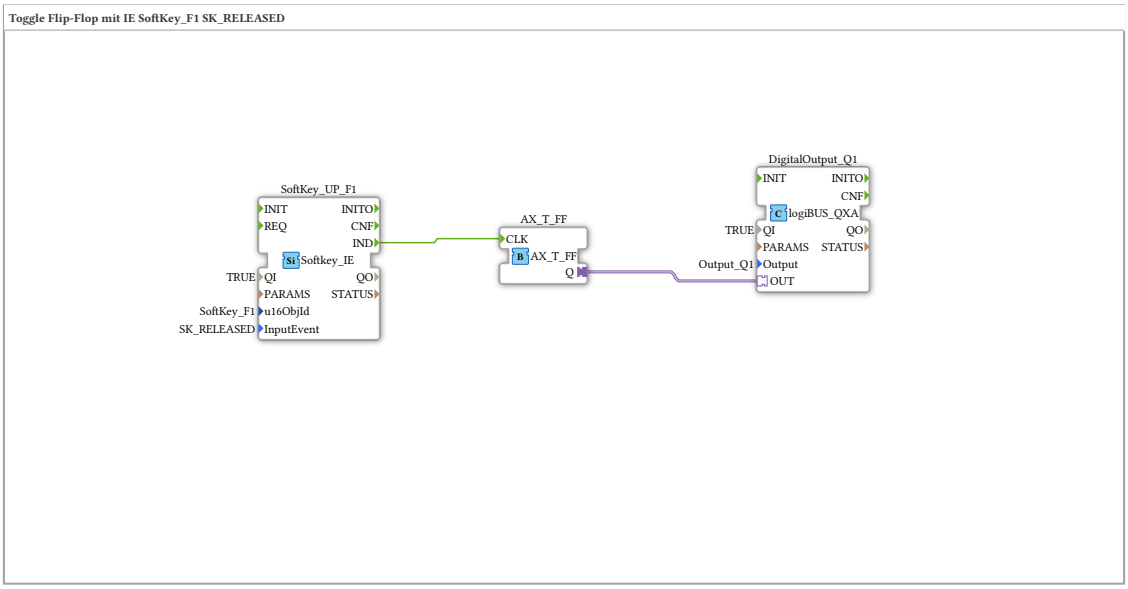
Funktionsweise

Im Gegensatz zu Softkeys, die fest auf dem Bildschirm sind, ist eine AUX-Funktion abstrakt. Der Nutzer muss am Terminal erst einen physischen Eingang (z.B. Taste auf dem Joystick) dieser Funktion („Funktion 2“) zuweisen (Teaching). Sobald das Mapping steht: Joystick-Taste drücken -> `Aux_IXA` wird TRUE -> Ausgang schaltet.

2.1.2.125 Uebung_010b2_AX: Toggle Flip-Flop mit IE SoftKey_F1 SK_RELEASED

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_010b2_AX`.

Podcast



- ISO 11783-6: Softkeys und das Virtual Terminal verstehen – Dein Schlüssel zur Landmaschinen-Mechatronik

---

Ziel der Übung

Verwendung von `Softkey_IE` (Event) anstelle von `Softkey_IXA` (Zustand).

Beschreibung und Komponenten

Die Subapplikation `Uebung_010b2_AX.SUB` nutzt einen Softkey, um ein Flip-Flop zu toggeln.

Funktionsbausteine (FBs)

- **`SoftKey_UP_F1`**: Typ `isobus::UT::io::Softkey::Softkey_IE`.
- **InputEvent**: `SK_RELEASED`.

Funktionsweise

Das Event wird gefeuert, wenn der Nutzer den Softkey **loslässt**. Dies ist das Standardverhalten für „Klick“-Interaktionen (ähnlich wie bei einer Maus). Das Flip-Flop wechselt bei jedem Loslassen den Zustand.

2.1.2.126 Uebung_010b3_AX: Toggle Flip-Flop mit IE AuxFunction2_X1

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_010b3_AX`.

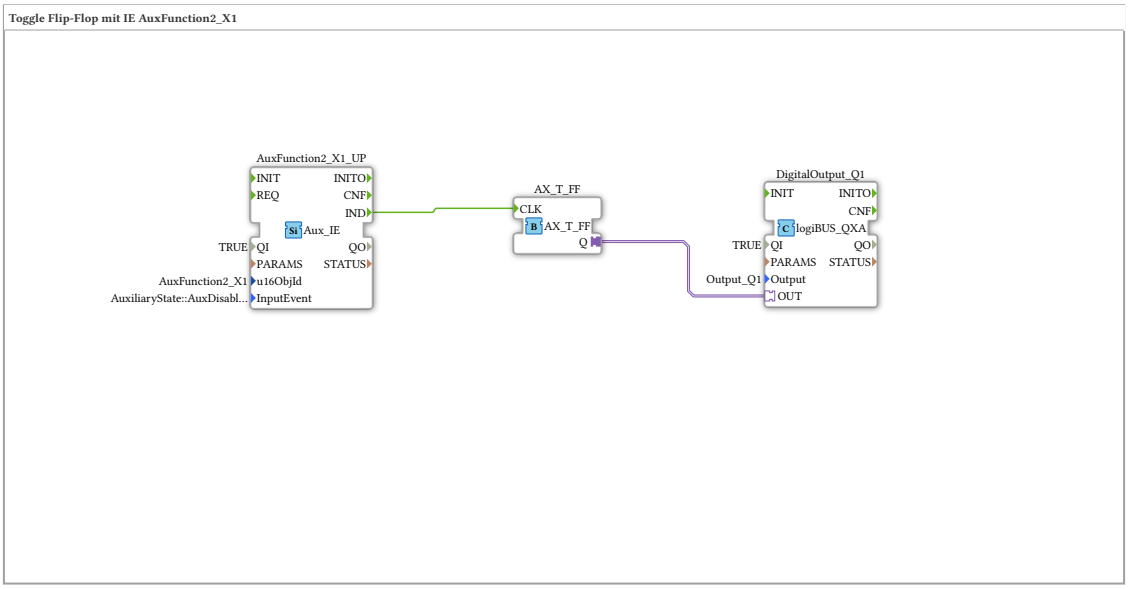
Ziel der Übung

Verwendung von `Aux_IE` (Event).

Beschreibung und Komponenten

Die Subapplikation `Uebung_010b3_AX.SUB` toggelt ein Flip-Flop über AUX.

Funktionsbausteine (FBs)



- **`AuxFunction2_X1_UP`**: Typ `isobus::UT::io::Auxiliary::IN::Aux_IE`.
- **InputEvent**: `AuxDisabled_START`.

Funktionsweise

Das Event-Namensschema bei AUX ist etwas speziell:

- `AuxDisabled`: Bedeutet, der Schalter ist „Aus“ (Offen).
- `AuxEnabled`: Bedeutet, der Schalter ist „Ein“ (Geschlossen).
- `_START`: Bedeutet Flanke (Übergang in diesen Zustand).

`AuxDisabled_START` bedeutet also: Der Übergang von „Enabled“ zu „Disabled“. Das entspricht dem **Loslassen** eines Tasters (`Falling Edge`). Das Flip-Flop schaltet also beim Loslassen der Joystick-Taste um.

2.1.2.127 Uebung_010b4_AX: SoftKey_F1-8 auf DigitalOutput_Q1-8





Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_010b4_AX`.

🎧 Podcast

- ISO 11783-6: Softkeys und das Virtual Terminal verstehen – Dein Schlüssel zur Landmaschinen-Mechatronik

---

Ziel der Übung

Effiziente Verbindung vieler Softkeys.

Beschreibung und Komponenten

Die Subapplikation `Uebung_010b4_AX.SUB` nutzt untypisierte Sub-Applikationen (`Uebung_010b4_sub_AX`), um 8 Softkeys auf 8 Ausgänge zu mappen.

Die Struktur ist analog zu `Uebung_003b2_AX`, nur mit ISOBUS-Softkeys als Quelle.

Funktionsweise

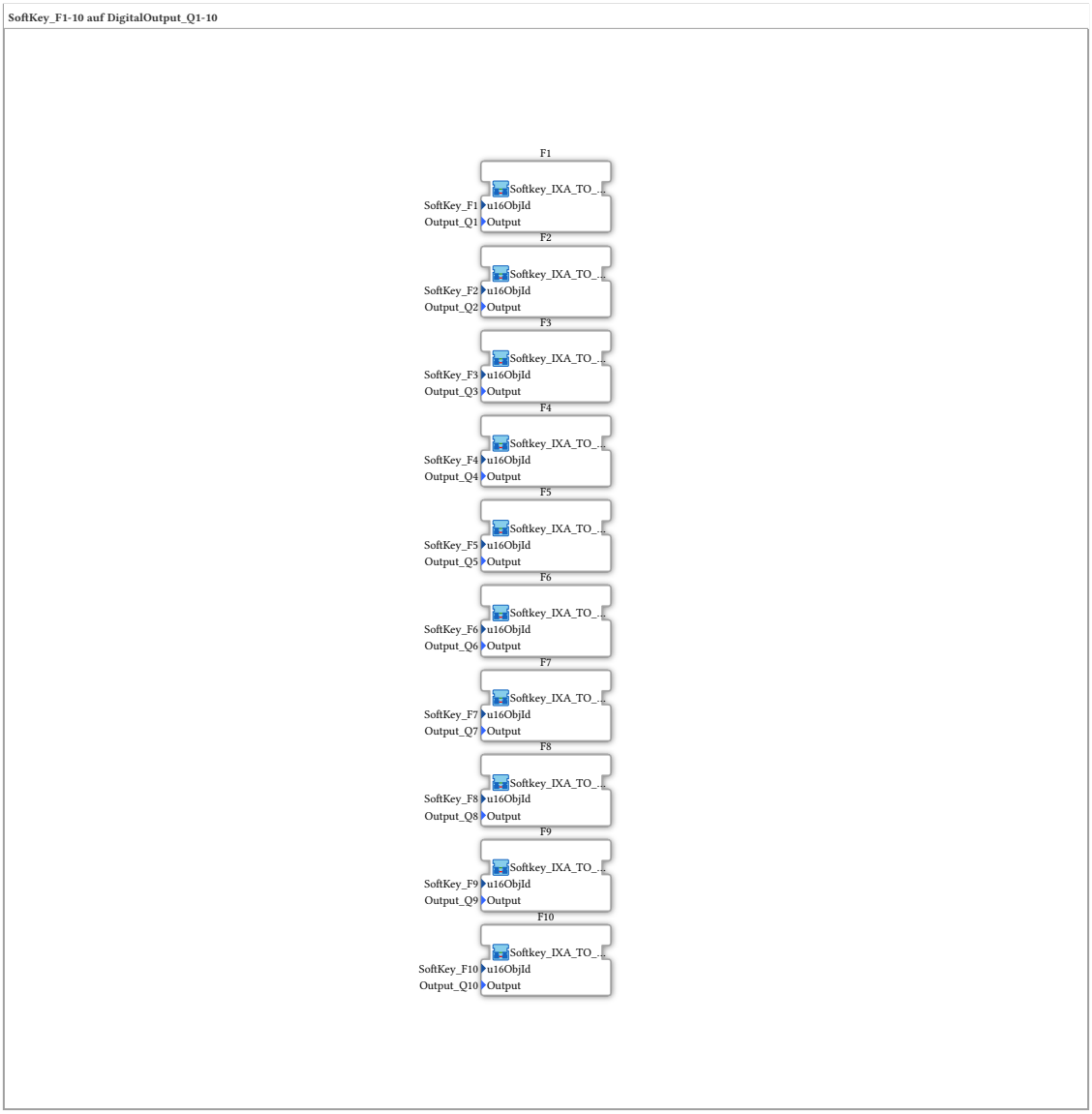
Jede SubApp kapselt die Verbindung `Softkey -> Ausgang`. Dies hält das Hauptdiagramm sauber.

2.1.2.128 Uebung_010b4_sub_AX

Hinweis

Diese Übung existiert in den aktuellen Trainingsquellen nicht mehr (entfernt, zusammengelegt oder umbenannt). Diese Seite bleibt nur als Platzhalter für bestehende Verweise erhalten. Die aktuellen Übungen stehen in der Übungsübersicht.

2.1.2.129 Uebung_010b5_AX: SoftKey_F1-10 auf DigitalOutput_Q1-10



NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_010b5_AX`.

🎧 Podcast

- ISO 11783-6: Softkeys und das Virtual Terminal verstehen – Dein Schlüssel zur Landmaschinen-Mechatronik

---

Ziel der Übung

Erweiterung auf 10 Kanäle.

Beschreibung

Identisch zu `Uebung_010b4_AX`, aber für 10 Softkeys (`F1` bis `F10`) auf 10 Ausgänge.

2.1.2.130 Uebung_010b5_sub_AX

Hinweis

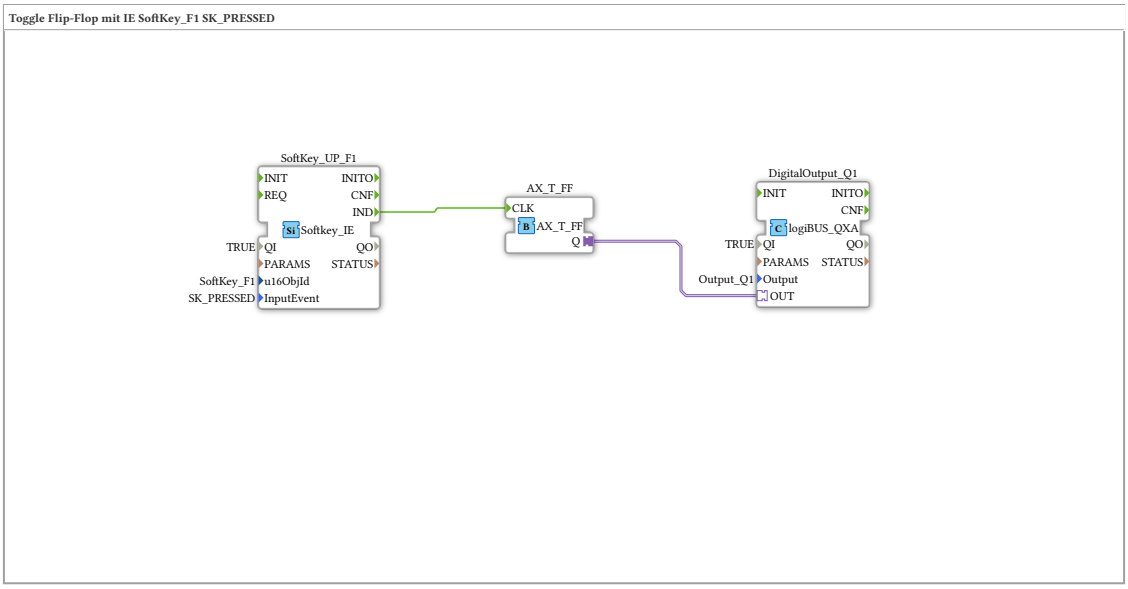
Diese Übung existiert in den aktuellen Trainingsquellen nicht mehr (entfernt, zusammengelegt oder umbenannt). Diese Seite bleibt nur als Platzhalter für bestehende Verweise erhalten. Die aktuellen Übungen stehen in der Übungsübersicht.

2.1.2.131 Uebung_010b6_AX: Toggle Flip-Flop mit IE SoftKey_F1 SK_PRESSED



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_010b6_AX`.

🎧 Podcast



- ISO 11783-6: Softkeys und das Virtual Terminal verstehen – Dein Schlüssel zur Landmaschinen-Mechatronik

---

Ziel der Übung

Reaktion beim Drücken.

Beschreibung

Verwendet das Event `SK_PRESSED`.

Funktionsweise

Das Flip-Flop schaltet bereits im Moment des Berührens („Touch Down“) um, nicht erst beim Loslassen. Das fühlt sich „schneller“ an, ist aber untypisch für Touch-Bedienoberflächen (wo man meist noch wegziehen kann, um abzubrechen).

2.1.2.132 Uebung_010b7_AX: Toggle Flip-Flop mit IE Button_A1 BT_RELEASED_UNLATCHED



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_010b7_AX`.

Ziel der Übung

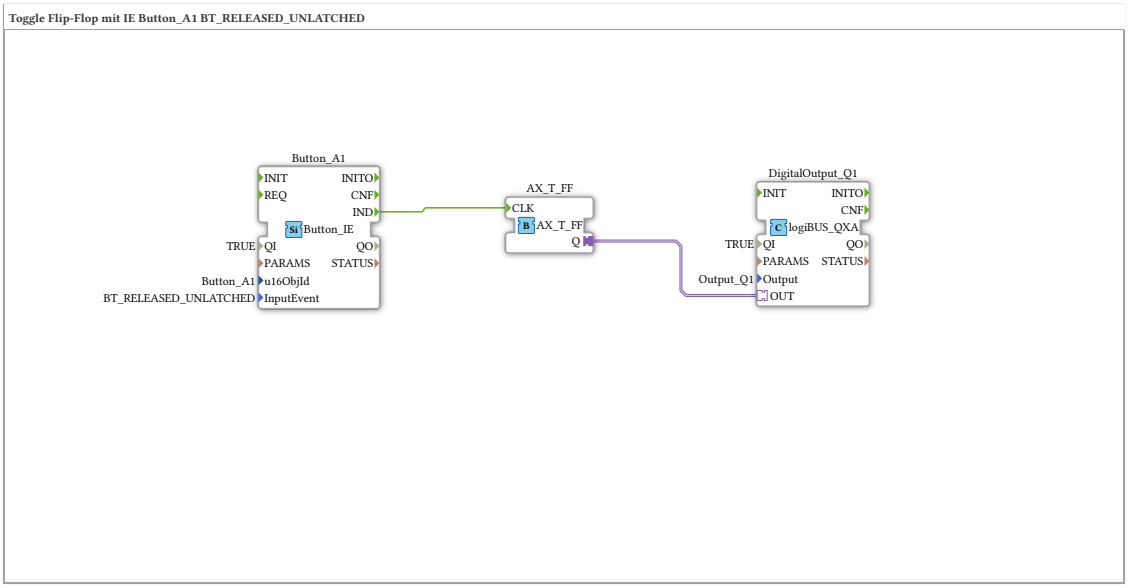
Events von Buttons.

Beschreibung

Nutzt `Button_A1` mit `BT_RELEASED_UNLATCHED`.

Funktionsweise

Feuert, wenn ein nicht-rastender Button losgelassen wird. Entspricht einem normalen Klick.



2.1.2.133 Uebung_010b8_AX: Toggle Flip-Flop mit IE Button_A1 BT_PRESSED_LATCHED



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_010b8_AX`.

Ziel der Übung

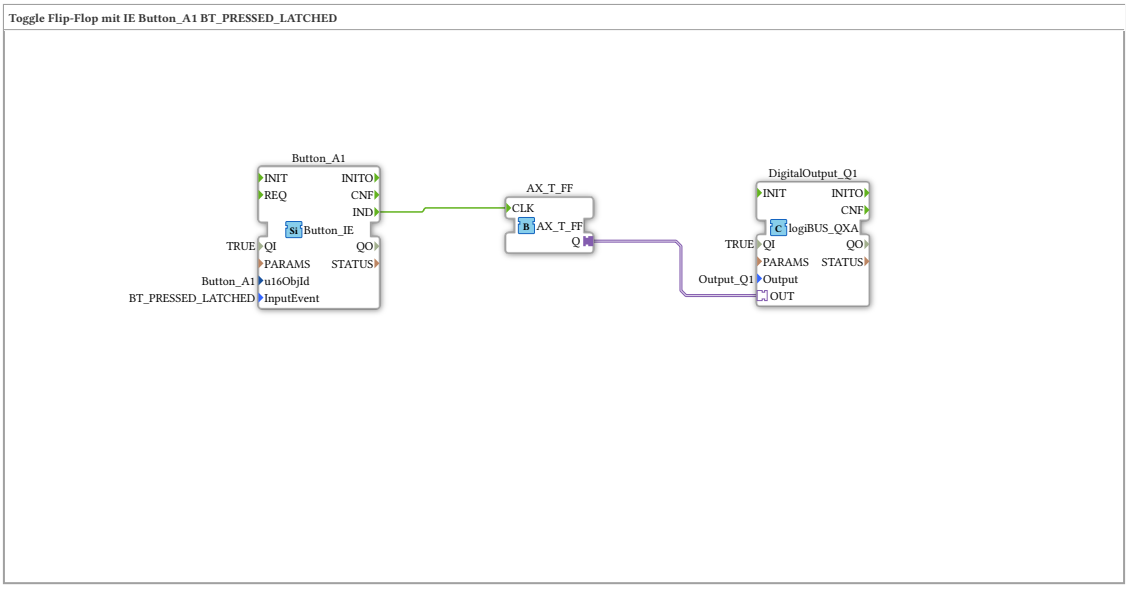
Events bei rastenden Buttons.

Beschreibung

Nutzt `Button_A1` mit `BT_PRESSED_LATCHED`.

Funktionsweise

Dies ist für Buttons gedacht, die visuell einrasten sollen. Das Event kommt, wenn der Button in den Zustand „Gedrückt/Latched“ übergeht.



2.1.2.134 Uebung_010b9_AX: Toggle Flip-Flop mit IE Button_A1 BT_STILL_HELD

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_010b9_AX`.

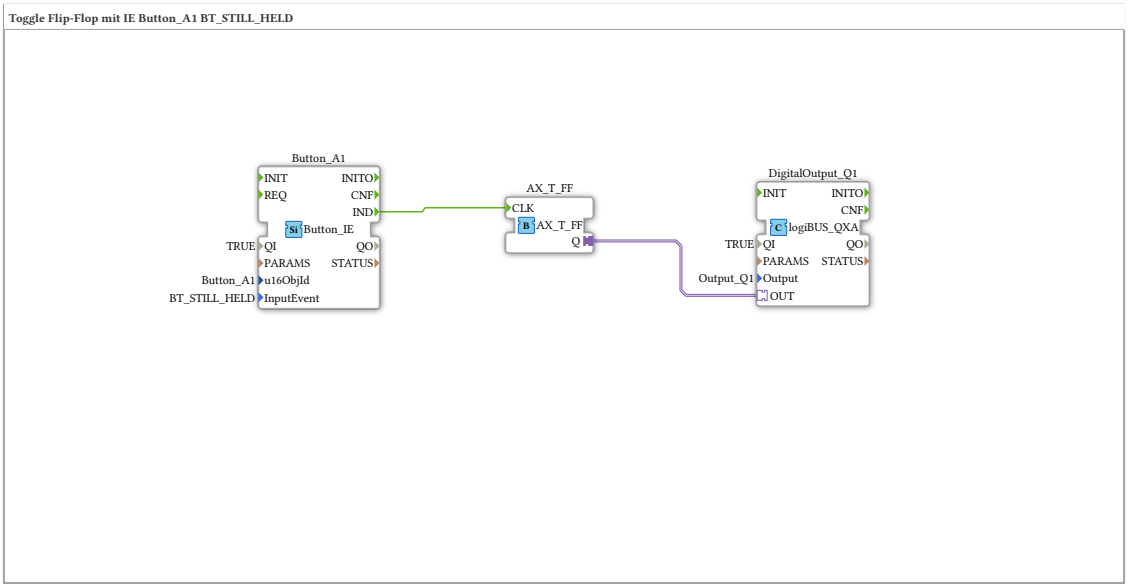
Ziel der Übung

Wiederholende Events.

Beschreibung

Nutzt `Button_A1` mit `BT_STILL_HELD`.

Funktionsweise



Wie im Kommentar beschrieben: *„BT_STILL_HELD wird alle 200ms wiederholt.“* Wenn der Nutzer den Button gedrückt hält, feuert der Baustein alle 200ms ein Event. Da dies an ein Toggle-Flip-Flop geht, blinkt der Ausgang (`Q1`) alle 200ms um (Periode 400ms), solange gedrückt wird.

2.1.2.135 Uebung_010bA2_AX: Toggle Flip-Flop mit IE AuxFunction2_X1 AuxEnabled



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_010bA2_AX`. Hier geht es um Feinheiten der AUX-Spezifikation.

Ziel der Übung

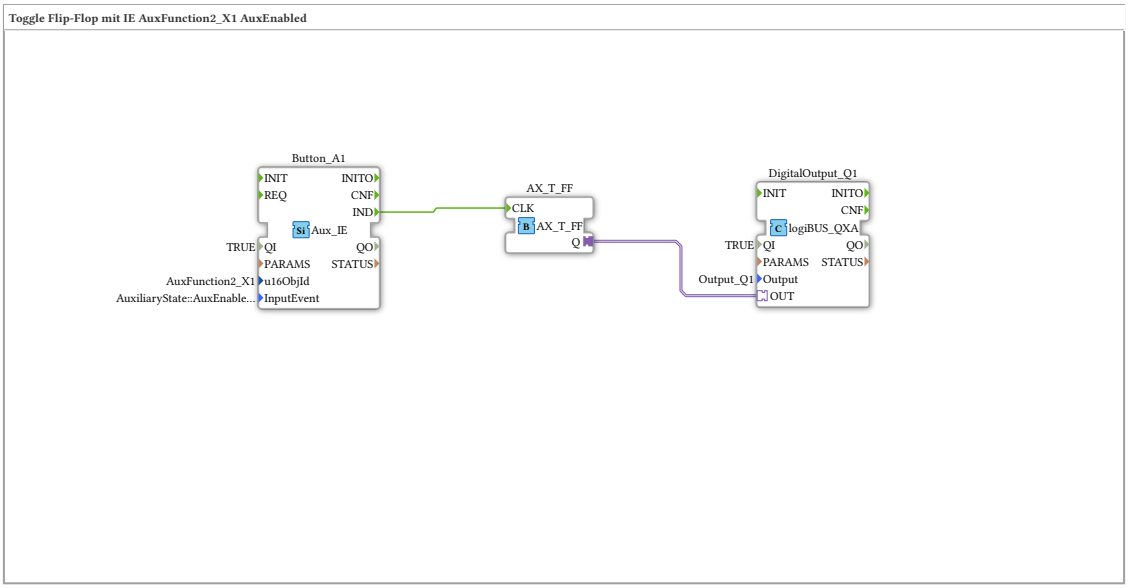
Verhalten von `AuxEnabled`.

Beschreibung

Nutzt `AuxFunction2_X1` mit `AuxEnabled`.

Funktionsweise

Kommentar erklärt den Unterschied je nach AUX-Typ (Bool_Latched=0 vs Bool_NonLatched=2).



Bei einem **Typ 2 (NonLatched)** (z.B. Taster am Joystick) wird `AuxEnabled` **einmal** gesendet beim Drücken. Bei einem **Typ 0 (Latched)** (z.B. Kippschalter) wird `AuxEnabled` **zyklisch wiederholt**, solange er an ist.

Da hier Typ 2 angenommen wird, verhält es sich wie ein normaler Klick.

2.1.2.136 Uebung_010bA3_AX: Toggle Flip-Flop mit IE AuxFunction2_X1 AuxHeld



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_010bA3_AX`.

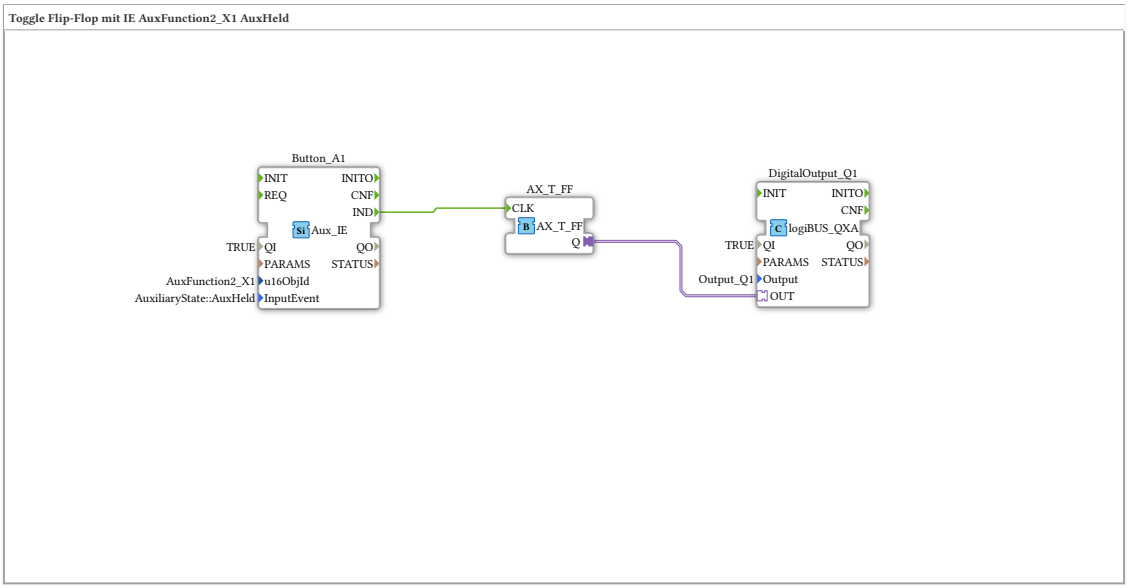
Ziel der Übung

Verhalten von `AuxHeld`.

Beschreibung

Nutzt `AuxFunction2_X1` mit `AuxHeld`.

Funktionsweise



Kommentar: *„AuxHeld wird wiederholt bei einem AUX Type 2. Ergibt einen Blinker.“* Ähnlich wie `BT_STILL_HELD`: Solange der Joystick-Knopf gehalten wird, kommen Events, und das Flip-Flop blinkt.

2.1.2.137 Uebung_010bA4_AX: Toggle Flip-Flop mit IE AuxFunction2_X1 AuxHeld_START

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_010bA4_AX`.

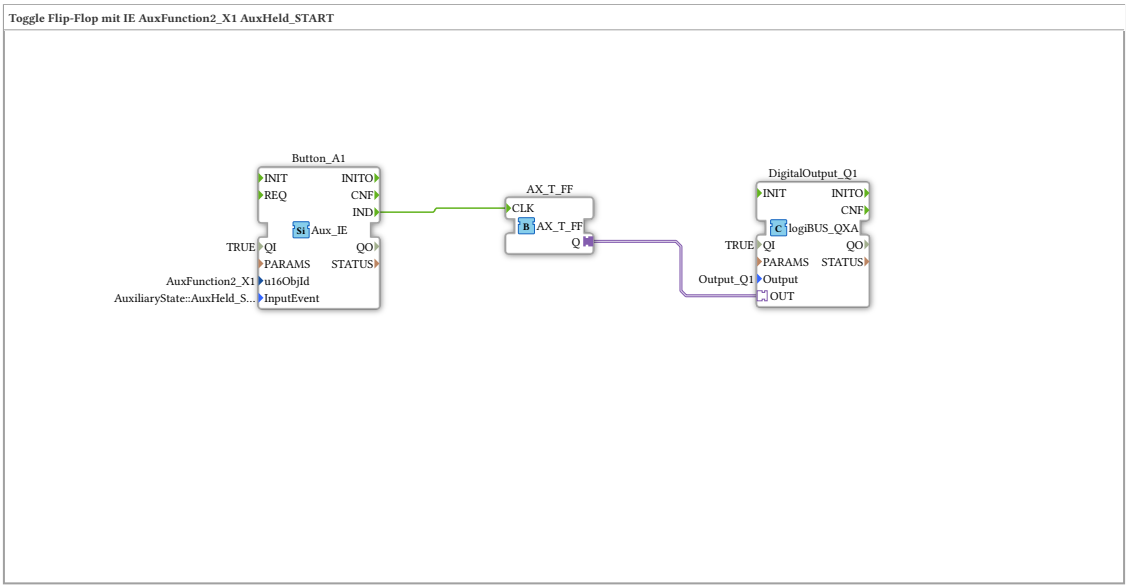
Ziel der Übung

Verhalten von `AuxHeld_START`.

Beschreibung

Nutzt `AuxFunction2_X1` mit `AuxHeld_START`.

Funktionsweise



Kommentar: *„AuxHeld_START wird nur einmal gesendet. Egal welcher Typ.“* Das ist das korrekte Event für „Long Press“ bei AUX.

2.1.2.138 Uebung_010bA_AX: Toggle Flip-Flop mit IE Button_A1 BT_STILL_HELD



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_010bA_AX`.

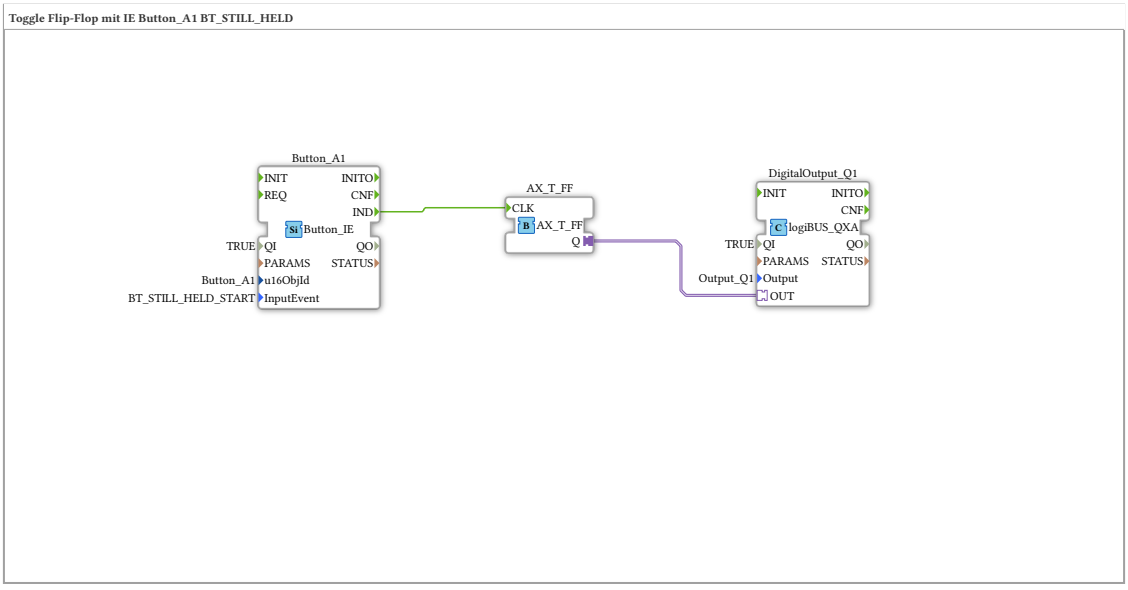
Ziel der Übung

Unterschied zu `STILL_HELD`.

Beschreibung

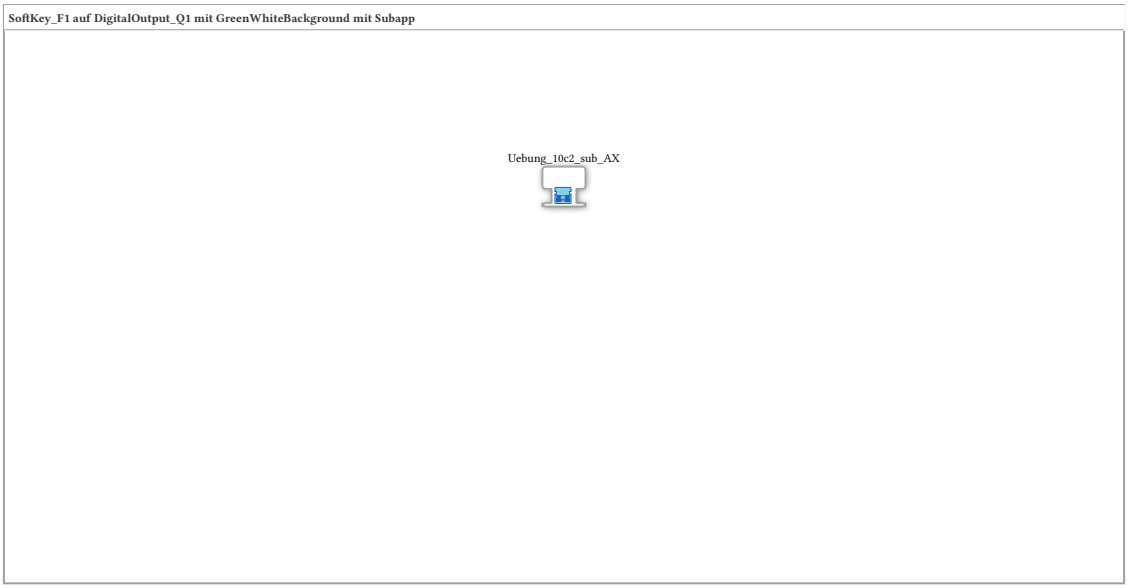
Nutzt `Button_A1` mit `BT_STILL_HELD_START`.

Funktionsweise



Kommentar: *„BT_STILL_HELD_START wird nicht wiederholt. Lange drücken ergibt 1 Event.“*Dies entspricht einem „Long Press“ Event für ISOBUS-Buttons. Es feuert genau einmal, wenn die Schwelle für „gehalten“ überschritten wird.

2.1.2.139 Uebung_010c2_AX: SoftKey_F1 auf DigitalOutput_Q1 mit GreenWhiteBackground mit Subapp



NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_010c2_AX`.

Ziel der Übung

Aufräumen des Diagramms.

Beschreibung

Die Logik aus `Uebung_010c_AX` (Input + Output + Feedback + Splitter) wird komplett in eine untypisierte Sub-Applikation `Uebung_10c2_sub_AX` verschoben.

Funktionsweise

Identisch zu vorher, aber im Hauptdiagramm sieht man nur noch „eine Box“. Dies macht deutlich, wie man komplexe UI-Logik verstecken kann.

2.1.2.140 Uebung_010c3_AX: SoftKey_F1 auf DigitalOutput_Q1 mit GreenWhiteBackground mit Typed Subapp



NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_010c3_AX`.

Ziel der Übung

Wiederverwendbarkeit durch Typisierung.

Beschreibung

Hier wurde aus der untypisierten SubApp ein echter Typ `Uebung_010c3_sub_AX` erstellt. Er hat Parameter `u16ObjId` (welcher Softkey?) und `Output` (welcher Ausgang?).

Vorteil

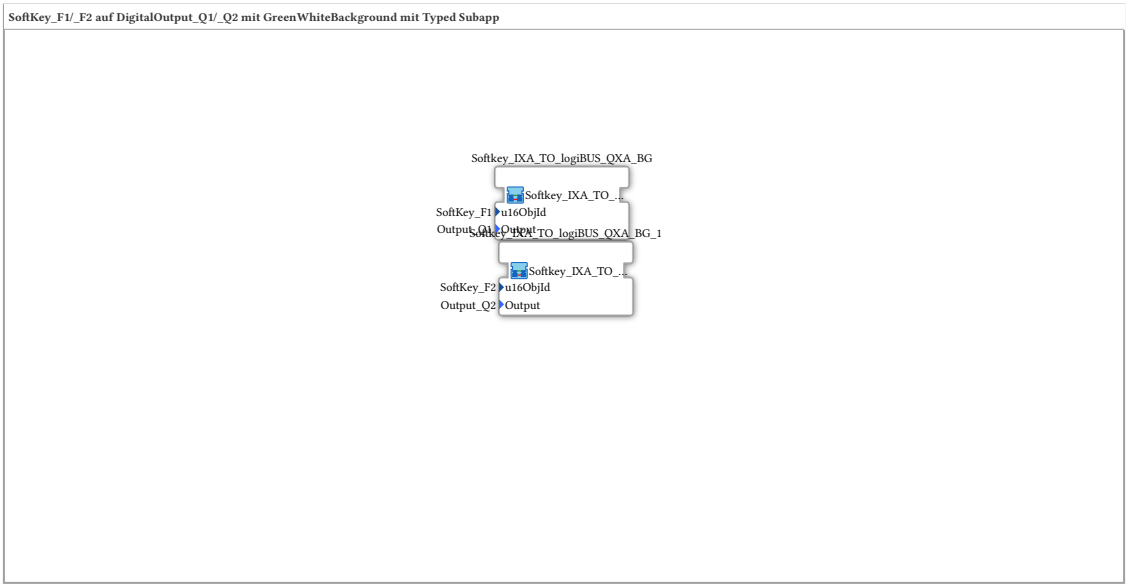
Man kann diesen „Smart Softkey“-Baustein nun beliebig oft in der Bibliothek verwenden.

2.1.2.141 Uebung_010c3_sub_AX

Hinweis

Diese Übung existiert nicht mehr als eigene Übung. Der Baustein wurde in die Bibliothek verschoben und heißt jetzt AuxIXA_TO_logiBUS_QXA_BG. Beschreibung, Schnittstelle und Netzwerk stehen dort.

2.1.2.142 Uebung_010c4_AX: SoftKey_F1/_F2 auf DigitalOutput_Q1/_Q2 mit GreenWhiteBackground mit Typed Subapp





Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_010c4_AX`.

🎧 Podcast

- ISO 11783-6: Softkeys und das Virtual Terminal verstehen – Dein Schlüssel zur Landmaschinen-Mechatronik

---

Ziel der Übung

Beweis der Wiederverwendbarkeit.

Beschreibung

Die Subapplikation `Uebung_010c4_AX.SUB` instanziiert den Typ `Uebung_010c4_sub_AX` (vermutlich identisch zu c3) zweimal.

- Instanz 1: F1 auf Q1.
- Instanz 2: F2 auf Q2.

Beide Softkeys haben nun automatisch die „Grün-bei-Druck“-Logik integriert.

2.1.2.143 Uebung_010c4_sub_AX

Hinweis

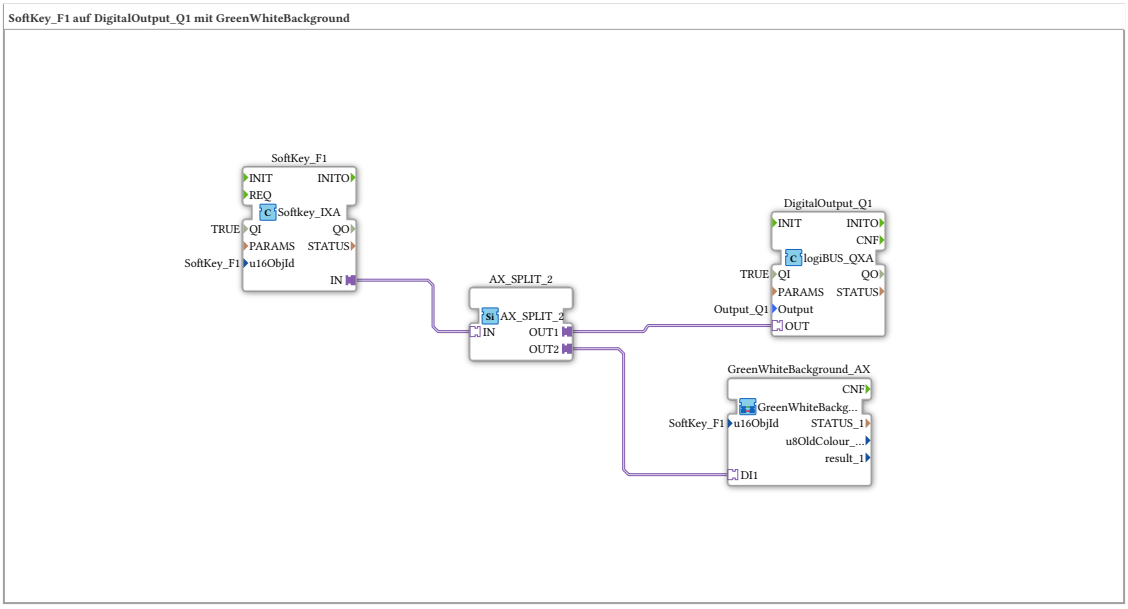
Diese Übung existiert in den aktuellen Trainingsquellen nicht mehr (entfernt, zusammengelegt oder umbenannt). Diese Seite bleibt nur als Platzhalter für bestehende Verweise erhalten. Die aktuellen Übungen stehen in der Übungsübersicht.

2.1.2.144 Uebung_010c_AX: SoftKey_F1 auf DigitalOutput_Q1 mit GreenWhiteBackground



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_010c_AX`. Bisher haben die Tasten nur geschaltet. Jetzt sollen sie auch leuchten.

🎧 Podcast



- ISO 11783-6: Softkeys und das Virtual Terminal verstehen – Dein Schlüssel zur Landmaschinen-Mechatronik

---

Ziel der Übung

Rückmeldung an den Bediener (Farbumschlag).

Beschreibung und Komponenten

Die Subapplikation `Uebung_010c_AX.SUB` erweitert die einfache Softkey-Schaltung um einen Feedback-Baustein.

Funktionsbausteine (FBs)

- **`SoftKey_F1`**: Eingabe.
- **`DigitalOutput_Q1`**: Ausgabe (Lampe).
- **`GreenWhiteBackground_AX`**: Eine SubApp aus der Bibliothek `MyLib::sys`. Diese steuert das Aussehen des Softkeys auf dem Terminal (Grün = Aktiv, Weiß = Inaktiv).
- **`AX_SPLIT_2`**: Verteilt das Signal vom Softkey sowohl an den Ausgang `Q1` als auch an den Feedback-Baustein.

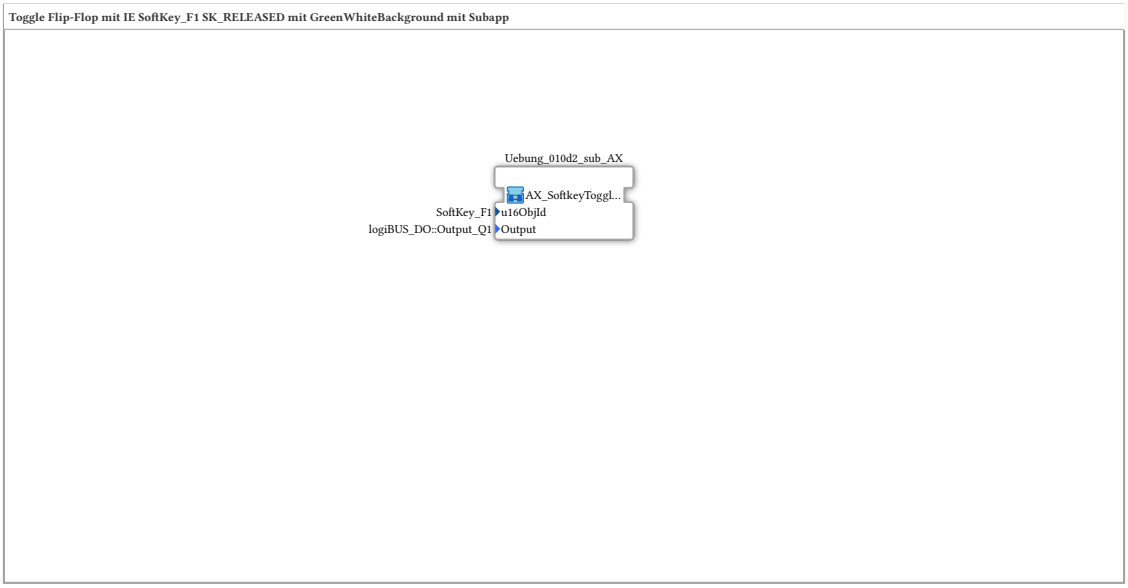
Funktionsweise

Wenn der Nutzer drückt, wird das Signal wahr.

1. Der physische Ausgang geht an.
2. Parallel dazu wird der Eingang `DI1` der Feedback-SubApp `TRUE`. Diese sendet ein ISOBUS-Kommando an das Terminal, um den Hintergrund des Softkeys `F1` auf Grün zu ändern.
3. Beim Loslassen wird der Ausgang aus und der Softkey wieder Weiß.

Dies gibt dem Nutzer direktes visuelles Feedback auf dem Touchscreen.

2.1.2.145 Uebung_010d2_AX: Toggle-Flipflop mit SoftKey_F1 und GreenWhiteBackground mit Typed Subapp



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_010d2_AX`.

Ziel der Übung

Wiederverwendbarkeit durch Typisierung.

Beschreibung

Hier wurde aus der untypisierten Toggle-Flipflop-Schaltung aus `Uebung_010d_AX` ein echter Typ `Uebung_010d2_sub_AX` erstellt. Er hat Parameter `u16ObjId` (welcher Softkey?) und `Output` (welcher Ausgang?).

Vorteil

Man kann diesen „Smart Toggle-Softkey“-Baustein nun beliebig oft in der Bibliothek verwenden.

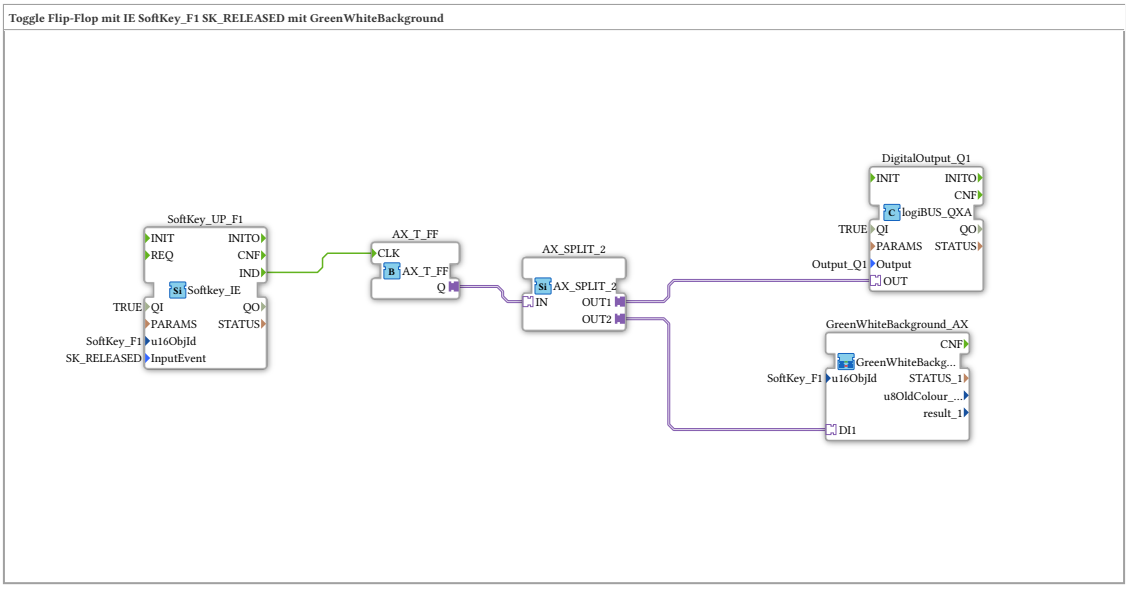
2.1.2.146 Uebung_010d2_sub_AX

Hinweis

Diese Übung existiert nicht mehr als eigene Übung. Der Baustein wurde in die Bibliothek verschoben und heißt jetzt AX_SoftkeyToggle_TO_QXA_BG. Beschreibung, Schnittstelle und Netzwerk stehen dort.

2.1.2.147 Uebung_010d_AX: Toggle-Flipflop mit SoftKey_F1 und GreenWhiteBackground

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_010d_AX`.



Ziel der Übung

Statt nur direkt durchzuschalten, soll der SoftKey `F1` jetzt einen Zustand umschalten (Toggle), bei jedem Loslassen der Taste.

Beschreibung und Komponenten

Die Subapplikation `Uebung_010d_AX.SUB` ersetzt die direkte Durchschaltung aus `Uebung_010c_AX` durch ein Toggle-Flipflop.

Funktionsbausteine (FBs)

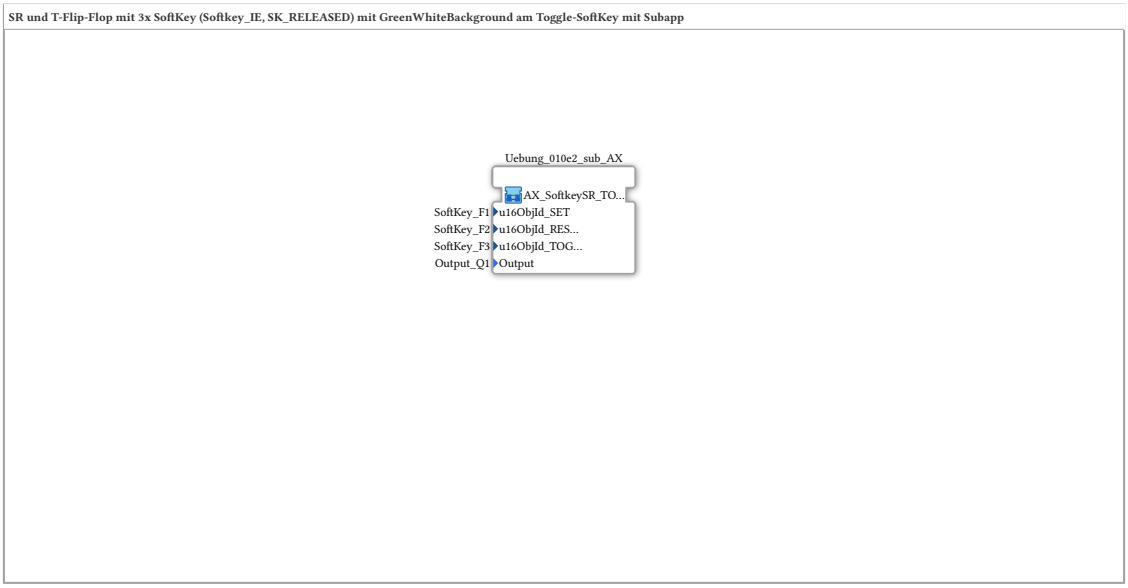
- **`SoftKey_UP_F1`**: `isobus::UT::io::Softkey::Softkey_IE`, ausgelöst durch das Ereignis `SK_RELEASED` (reagiert erst beim Loslassen der Taste, nicht beim Drücken).
- **`AX_T_FF`**: Toggle-Flipflop-Adapter. Jedes Taktereignis an `CLK` kehrt den Ausgangszustand `Q` um.
- **`AX_SPLIT_2`**: Verteilt das Adaptersignal von `AX_T_FF.Q` sowohl an den Ausgang `Q1` als auch an den Feedback-Baustein.
- **`DigitalOutput_Q1`**: Ausgabe (Lampe), `Output_Q1`.
- **`GreenWhiteBackground_AX`**: SubApp aus `MyLib::sys`, steuert das Aussehen des Softkeys `F1` auf dem Terminal (Grün = Aktiv, Weiß = Inaktiv).

Funktionsweise

1. Der Nutzer lässt den SoftKey `F1` los; `SoftKey_UP_F1` löst das Ereignis `IND` aus.
2. `AX_T_FF` kehrt seinen internen Zustand `Q` um (aus AN wird AUS, aus AUS wird AN).
3. `AX_SPLIT_2` verteilt den neuen Zustand an `DigitalOutput_Q1` (physischer Ausgang) und `GreenWhiteBackground_AX` (Softkey-Hintergrundfarbe).

Im Gegensatz zu `Uebung_010c_AX`, wo der Ausgang nur während des Tastendrucks aktiv ist, bleibt der Zustand hier bis zum nächsten Loslassen der Taste erhalten.

2.1.2.148 Uebung_010e2_AX: SR+Toggle-Flipflop mit 3x SoftKey und GreenWhiteBackground mit Typed Subapp



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_010e2_AX`.

Ziel der Übung

Wiederverwendbarkeit durch Typisierung.

Beschreibung

Hier wurde aus der untypisierten SR+Toggle-Flipflop-Schaltung aus `Uebung_010e_AX` ein echter Typ `Uebung_010e2_sub_AX` erstellt. Er hat die Parameter `u16ObjId_SET`, `u16ObjId_RESET`, `u16ObjId_TOGGLE` (welcher SoftKey löst welche Aktion aus?) und `Output` (welcher Ausgang?).

Vorteil

Man kann diesen „Smart SR+Toggle-Softkey“-Baustein nun beliebig oft in der Bibliothek verwenden.

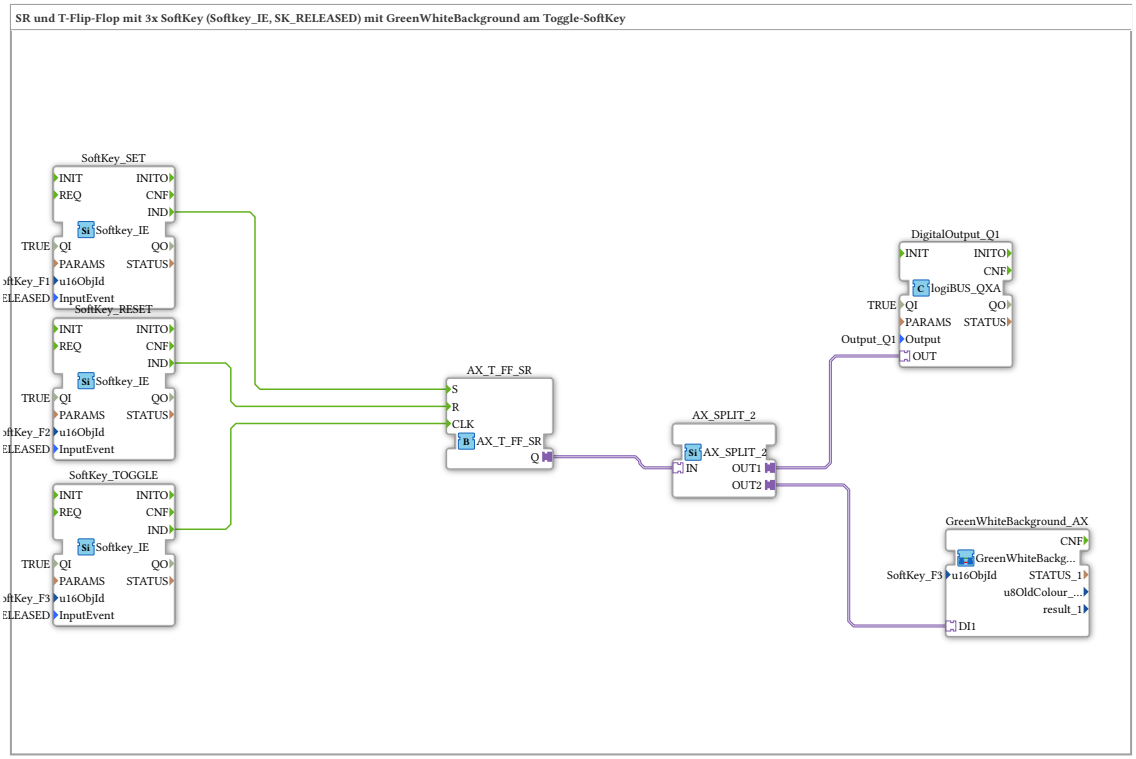
2.1.2.149 Uebung_010e2_sub_AX

Hinweis

Diese Übung existiert nicht mehr als eigene Übung. Der Baustein wurde in die Bibliothek verschoben und heißt jetzt AX_SoftkeySR_TO_QXA_BG. Beschreibung, Schnittstelle und Netzwerk stehen dort.

2.1.2.150 Uebung_010e_AX: SR+Toggle-Flipflop mit 3x SoftKey und GreenWhiteBackground

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_010e_AX`.



Ziel der Übung

Ein Ausgang soll über drei getrennte SoftKeys gesteuert werden können: gezielt Setzen, gezielt Rücksetzen, oder Umschalten (Toggle).

Beschreibung und Komponenten

Die Subapplikation `Uebung_010e_AX.SUB` erweitert das reine Toggle-Flipflop aus `Uebung_010d_AX` um ein kombiniertes SR+Toggle-Flipflop mit drei SoftKeys.

Funktionsbausteine (FBs)

- **`SoftKey_SET`**: `isobus::UT::io::Softkey::Softkey_IE` auf `SoftKey_F1`, Ereignis `SK_RELEASED`.
- **`SoftKey_RESET`**: dasselbe auf `SoftKey_F2`.
- **`SoftKey_TOGGLE`**: dasselbe auf `SoftKey_F3`.
- **`AX_T_FF_SR`**: SR+Toggle-Flipflop-Adapter mit den Eingängen `S` (Set), `R` (Reset) und `CLK` (Toggle).
- **`AX_SPLIT_2`**: Verteilt das Adaptersignal von `AX_T_FF_SR.Q` sowohl an den Ausgang `Q1` als auch an den Feedback-Baustein.
- **`DigitalOutput_Q1`**: Ausgabe (Lampe), `Output_Q1`.
- **`GreenWhiteBackground_AX`**: SubApp aus `MyLib::sys`, an `SoftKey_F3` (den Toggle-SoftKey) gebunden — nur dieser zeigt die aktuelle Hintergrundfarbe.

Funktionsweise

1. `SoftKey_SET.IND` ⟶ `AX_T_FF_SR.S`: setzt den Zustand `Q` fest auf AN, unabhängig vom vorherigen Zustand.
2. `SoftKey_RESET.IND` ⟶ `AX_T_FF_SR.R`: setzt den Zustand `Q` fest auf AUS.
3. `SoftKey_TOGGLE.IND` ⟶ `AX_T_FF_SR.CLK`: kehrt den aktuellen Zustand `Q` um.
4. `AX_SPLIT_2` verteilt den neuen Zustand an `DigitalOutput_Q1` (physischer Ausgang) und `GreenWhiteBackground_AX` (Hintergrundfarbe des Toggle-SoftKeys `F3`).

Set und Reset wirken also unabhängig vom aktuellen Zustand, während Toggle ihn umkehrt — alle drei SoftKeys steuern denselben internen Flipflop-Zustand.

2.1.2.151 Uebung_010f2_AX: AuxFunction2_X1 auf DigitalOutput_Q1 mit GreenWhite-Background mit Subapp



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung löst den in `Uebung_010f_AX` genannten Nachteil auf, dass die Objekt-ID `AuxFunction2_X1` an zwei Stellen (Aux-FB und `GreenWhiteBackground2_AX`) einzeln eingetragen werden muss – genau wie `Uebung_010c2_AX` es für `Uebung_010c_AX` tut. Die gesamte Netzwerkstruktur besteht auf oberster Ebene nur noch aus einer einzigen SubApp-Instanz eines Bausteins aus der Bibliothek `MyLib_AX`, der die komplette Logik kapselt.

Verwendete Funktionsbausteine (FBs)

Auf oberster Ebene der SubApp befindet sich kein einzelner FB, sondern ausschließlich die unten beschriebene SubApp-Instanz.

- **Uebung_010f2_sub_AX**: Gekapselte Logik aus Aux-Eingang, Splitter, Ausgang und Hintergrundfarbe
  - **Typ**: `MyLib::sys::AuxIXA_TO_logiBUS_QXA_BG` (SubApp-Instanz)
  - **Parameter**: `u16ObjIdA = AuxFunction2_X1` (Objekt-ID des Aux-Objekts), `Output = logiBUS_DO::Output_Q1` (Identität des physischen Ausgangs)
  - **Erklärung**: Die SubApp hat genau zwei Eingänge, `u16ObjIdA` und `Output`. Intern enthält sie vier Bausteine: `AuxFunction2_X1` (`isobus::UT::io::Auxiliary::IN::Aux_IXA`, `QI = TRUE`), `AX_SPLIT_2` (`adapter::events::unidirectional::AX_SPLIT_2`), `DigitalOutput_Q1` (`logiBUS::io::DQ::logiBUS_QXA`, `QI = TRUE`) und `GreenWhiteBackground2_AX` (`MyLib::sys::GreenWhiteBackground2_AX`).

Der Import `Uebungen::const::UT::DefaultPool::AuxFunction2_X1` liefert die Objekt-ID-Konstante, `logiBUS_DO` den Namensraum für die Ausgangskennung.

Programmablauf und Verbindungen

1. Beim Instanziieren der SubApp wird `Uebung_010f2_sub_AX` mit den beiden Parametern `u16ObjIdA` (Objekt-ID des Aux-Objekts) und `Output` (Ziel-Ausgang) versehen.
2. Innerhalb der SubApp verteilen (unsichtbare) DataConnections `u16ObjIdA` an `AuxFunction2_X1.u16ObjId` und an `GreenWhiteBackground2_AX.u16ObjIdA`; eine weitere verteilt `Output` an `DigitalOutput_Q1.Output`. Die Objekt-ID muss dadurch nur noch einmal angegeben werden.
3. Der Signalfluss innerhalb der SubApp (AdapterConnections): `AuxFunction2_X1.IN` ⟶ `AX_SPLIT_2.IN`, `AX_SPLIT_2.OUT1` ⟶ `DigitalOutput_Q1.OUT` (der physische Ausgang folgt dem Aux-Zustand) und `AX_SPLIT_2.OUT2` ⟶ `GreenWhiteBackground2_AX.DI1` (Hintergrundfarbe auf dem VT-Bildschirm und am Aux-Handle).

Die AUX-Farbe geht weiterhin sowohl an den VT-Bildschirm als auch an das Aux-Handle – das ist notwendig und wird hier nicht wegoptimiert.

Zusammenfassung

`Uebung_010f2_AX` demonstriert, wie sich eine mehrfach benötigte Objekt-ID durch Kapselung in einen eigenen SubApp-Typ (hier der Bibliotheksbaustein `MyLib::sys::AuxIXA_TO_logiBUS_QXA_BG`) auf einen einzigen Eintragspunkt reduzieren lässt. Funktional ist die Übung identisch zu `Uebung_010f_AX`; der Unterschied liegt ausschließlich in der Wiederverwendbarkeit und Wartbarkeit der Verdrahtung.

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

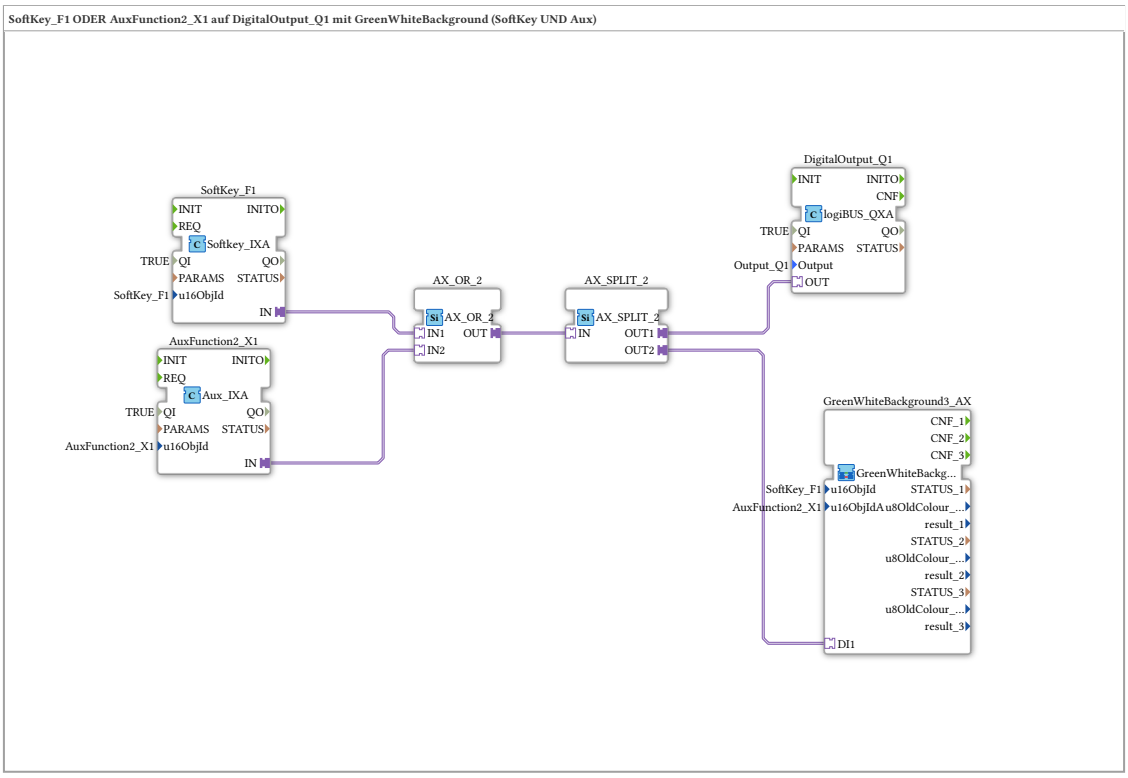
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.1.2.152 Uebung_010f2_sub_AX

Hinweis

Diese Übung existiert nicht mehr als eigene Übung. Der Baustein wurde in die Bibliothek verschoben und heißt jetzt AuxIXA_TO_logiBUS_QXA_BG. Beschreibung, Schnittstelle und Netzwerk stehen dort.

2.1.2.153 Uebung_010f3_AX: SoftKey_F1 ODER AuxFunction2_X1 auf DigitalOutput_Q1 mit GreenWhiteBackground



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung kombiniert `Uebung_010c_AX` (SoftKey mit Hintergrundfarbe) und `Uebung_010f_AX` (Aux mit Hintergrundfarbe) auf einen gemeinsamen Ausgang: dieselbe Funktion ist sowohl über den VT-SoftKey als auch über den physischen Aux-Taster/Joystick bedienbar. Sie zeigt den korrekten, idiomatischen Lösungsweg für diesen kombinierten Fall – als Kontrast dazu siehe die absichtliche Negativlösung `Uebung_010f4_AX`.

Verwendete Funktionsbausteine (FBs)

- **SoftKey_F1**: VT-SoftKey-Eingang (Typ: `isobus::UT::io::Softkey::Softkey_IXA`)
  - **Parameter**: `QI = TRUE`, `u16ObjId = SoftKey_F1`

  - **Erklärung**: Liefert den booleschen Zustand des SoftKeys als AX-Adapterausgang `IN`.
- **AuxFunction2_X1**: Auxiliary-Eingang (Typ: `isobus::UT::io::Auxiliary::IN::Aux_IXA`)
  - **Parameter**: `QI = TRUE`, `u16ObjId = AuxFunction2_X1`
  - **Erklärung**: Liefert den booleschen Zustand des zugewiesenen physischen Aux-Tasters/Joysticks als AX-Adapterausgang `IN`.
- **AX_OR_2**: Adapter-ODER-Verknüpfung (Typ: `adapter::booleanOperators::AX_OR_2`)
  - **Parameter**: keine
  - **Erklärung**: Verknüpft die beiden AX-Signale von SoftKey und Aux-Taster per ODER zu einem gemeinsamen Signal für `DigitalOutput_Q1`.
- **AX_SPLIT_2**: Adapter-Signalverteiler (Typ: `adapter::events::unidirectional::AX_SPLIT_2`)
  - **Parameter**: keine
  - **Erklärung**: Verteilt das ODER-verknüpfte Signal auf zwei Ziele: den physischen Ausgang und die kombinierte Hintergrundfarbe.
- **DigitalOutput_Q1**: logiBUS Digitalausgang (Typ: `logiBUS::io::DQ::logiBUS_QXA`)
  - **Parameter**: `QI = TRUE`, `Output = Output_Q1`
  - **Erklärung**: Schaltet den physischen Ausgang `Q1`, sobald SoftKey oder Aux-Taster aktiv sind.

Sub-Bausteine: GreenWhiteBackground3_AX

- **GreenWhiteBackground3_AX** (Typ: `MyLib::sys::GreenWhiteBackground3_AX`)
  - **Parameter**: `u16ObjId = SoftKey_F1`, `u16ObjIdA = AuxFunction2_X1`
  - **Erklärung**: Nimmt genau einen normalen VT-Objekt-Parameter (`u16ObjId` = SoftKey) UND einen Aux-Objekt-Parameter (`u16ObjIdA` = Aux) gleichzeitig entgegen und setzt beide Hintergründe aus einem einzigen `DI1`-Eingang. Intern sendet er drei Kommandos (`Q_BackgroundColour` für den SoftKey, `Q_BackgroundColour` UND `Q_BackgroundColourAux` für die Aux-Funktion), nach außen genügt daher ein Split auf zwei statt drei Ziele.

Programmablauf und Verbindungen

1. `SoftKey_F1.IN` ⟶ `AX_OR_2.IN1` und `AuxFunction2_X1.IN` ⟶ `AX_OR_2.IN2`: beide Bedienelemente liefern je ein AX-Signal an die ODER-Verknüpfung.
2. `AX_OR_2.OUT` ⟶ `AX_SPLIT_2.IN`: das kombinierte Signal wird auf zwei Verbraucher dupliziert.
3. `AX_SPLIT_2.OUT1` ⟶ `DigitalOutput_Q1.OUT`: der physische Ausgang `Q1` schaltet EIN, sobald SoftKey ODER Aux-Taster aktiv ist.
4. `AX_SPLIT_2.OUT2` ⟶ `GreenWhiteBackground3_AX.DI1`: derselbe kombinierte Zustand steuert gleichzeitig beide Hintergrundfarben (SoftKey auf dem VT, Aux auf VT und Aux-Handle).

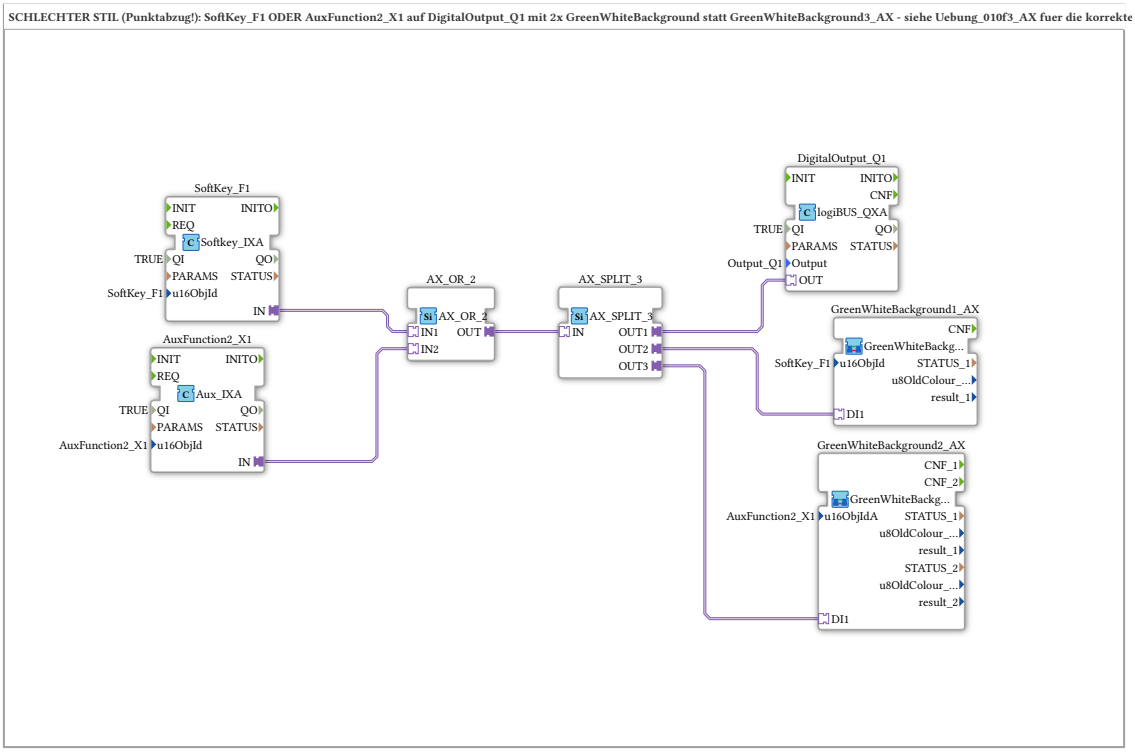
Zusammenfassung

Die Übung zeigt, wie zwei unterschiedliche Bedienelement-Typen (SoftKey und Aux) über eine gemeinsame ODER-Verknüpfung denselben Ausgang steuern, und wie der dafür vorgesehene Baustein `GreenWhiteBackground3_AX` beide Rückmeldungen aus nur einem `DI1`-Eingang und mit nur einem `AX_SPLIT_2` bedient – die ressourcenschonende, korrekte Lösung im Vergleich zur bewusst fehlerhaften `Uebung_010f4_AX`.

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

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

2.1.2.154 Uebung_010f4_AX: SCHLECHTER STIL (Punktabzug!) — SoftKey_F1 ODER AuxFunction2_X1 auf DigitalOutput_Q1 mit 2x GreenWhiteBackground



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Datei ist absichtlich als Negativbeispiel stehen gelassen (im Quellcode ausdrücklich als „SCHLECHTER STIL – Punktabzug!“ gekennzeichnet). Sie erreicht funktional dasselbe wie Uebung_010f3_AX (SoftKey ODER Aux-Taster schalten gemeinsam `Q1`), tut dies aber mit unnötig aufgeblähter Verdrahtung: statt des dafür vorgesehenen Bausteins `GreenWhiteBackground3_AX` werden zwei separate Instanzen `GreenWhiteBackground1_AX` und `GreenWhiteBackground2_AX` verwendet. Diese Seite dokumentiert bewusst den Fehler, nicht eine empfohlene Lösung – für die korrekte Umsetzung siehe Uebung_010f3_AX.

Verwendete Funktionsbausteine (FBs)

- **SoftKey_F1**: VT-SoftKey-Eingang
  - **Typ**: `isobus::UT::io::Softkey::Softkey_IXA`
  - **Parameter**: `QI = TRUE`, `u16ObjId = SoftKey_F1`
  - **Erklärung**: Liefert den booleschen Zustand des SoftKeys als AX-Adapterausgang `IN`.
- **AuxFunction2_X1**: Auxiliary-Eingang
  - **Typ**: `isobus::UT::io::Auxiliary::IN::Aux_IXA`
  - **Parameter**: `QI = TRUE`, `u16ObjId = AuxFunction2_X1`
  - **Erklärung**: Liefert den booleschen Zustand des zugewiesenen physischen Aux-Tasters/Joysticks als AX-Adapterausgang `IN`.
- **AX_OR_2**: Adapter-ODER-Verknüpfung
  - **Typ**: `adapter::booleanOperators::AX_OR_2`
  - **Parameter**: keine
  - **Erklärung**: Verknüpft die beiden AX-Signale von SoftKey und Aux-Taster per ODER zu einem gemeinsamen Signal.
- **AX_SPLIT_3**: Adapter-Signalverteiler auf drei Ziele
  - **Typ**: `adapter::events::unidirectional::AX_SPLIT_3`
  - **Parameter**: keine

  - **Erklärung**: Verteilt das ODER-verknüpfte Signal auf **drei** statt zwei Ziele, weil hier zwei getrennte Hintergrundfarben-Bausteine statt eines kombinierten bedient werden müssen – der eigentliche Konstruktionsfehler dieser Übung.
- **DigitalOutput_Q1**: logiBUS Digitalausgang
  - **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
  - **Parameter**: `QI = TRUE`, `Output = Output_Q1`
  - **Erklärung**: Schaltet den physischen Ausgang `Q1`, sobald SoftKey oder Aux-Taster aktiv sind.
- **GreenWhiteBackground1_AX**: Hintergrundfarbe für ein normales VT-Objekt
  - **Typ**: `MyLib::sys::GreenWhiteBackground1_AX` (SubApp-Instanz)
  - **Parameter**: `u16ObjId = SoftKey_F1`
  - **Erklärung**: Setzt die Hintergrundfarbe des SoftKey-Objekts auf dem VT (TRUE → Grün, FALSE → Weiß) über ein einzelnes VT-Kommando. Wird hier als **separate** Instanz benötigt, weil kein kombinierter Baustein verwendet wurde.
- **GreenWhiteBackground2_AX**: Hintergrundfarbe für ein Auxiliary-Function-Objekt
  - **Typ**: `MyLib::sys::GreenWhiteBackground2_AX` (SubApp-Instanz)
  - **Parameter**: `u16ObjIdA = AuxFunction2_X1`
  - **Erklärung**: Setzt die Hintergrundfarbe des Aux-Objekts (TRUE → Grün, FALSE → Weiß) über dieselbe `u16ObjIdA` an zwei Empfänger (VT-Bildschirm und Aux-Handle). Ebenfalls als **separate** Instanz statt in `GreenWhiteBackground3_AX` zusammengefasst.

Programmablauf und Verbindungen

1. `SoftKey_F1.IN` → `AX_OR_2.IN1` und `AuxFunction2_X1.IN` → `AX_OR_2.IN2` (AdapterConnections): beide Bedienelemente liefern je ein AX-Signal an die ODER-Verknüpfung.
2. `AX_OR_2.OUT` → `AX_SPLIT_3.IN`: das kombinierte Signal wird auf **drei** statt zwei Verbraucher verteilt.
3. `AX_SPLIT_3.OUT1` → `DigitalOutput_Q1.OUT`: der physische Ausgang `Q1` schaltet EIN, sobald SoftKey ODER Aux-Taster aktiv ist.
4. `AX_SPLIT_3.OUT2` → `GreenWhiteBackground1_AX.DI1`: separate Ansteuerung der SoftKey-Hintergrundfarbe.
5. `AX_SPLIT_3.OUT3` → `GreenWhiteBackground2_AX.DI1`: separate Ansteuerung der Aux-Hintergrundfarbe.

Was hier falsch gemacht wurde: Durch die Verwendung von zwei einzelnen `GreenWhiteBackground1_AX`/`GreenWhiteBackground2_AX`-Instanzen statt des dafür gebauten kombinierten Bausteins `GreenWhiteBackground3_AX` (nimmt `u16ObjId` UND `u16ObjIdA` gleichzeitig und braucht nur einen `DI1`-Eingang) war ein `AX_SPLIT_3` (3 Ziele) statt eines simplen `AX_SPLIT_2` (2 Ziele) nötig. Die Farbauswahl-Logik (intern `AX_SEL` Weiß/Grün) läuft dadurch doppelt statt einmal – unnötiger Ressourcenverbrauch und unnötig große Diagramme, ohne jeden funktionalen Vorteil gegenüber Uebung_010f3_AX.

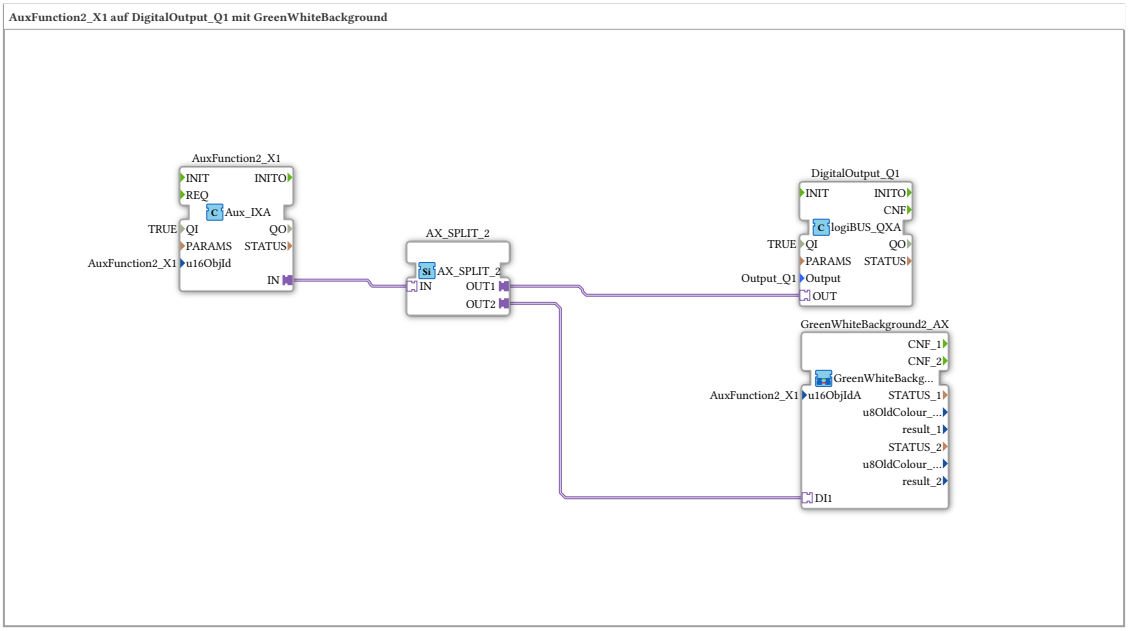
Zusammenfassung

Diese Übung ist bewusst kein Vorbild, sondern eine Lehrbeispiel-Warnung: Bevor man für einen kombinierten Fall (normales VT-Objekt + Aux-Objekt) mehrere generische Bausteine einzeln zusammenschraubt, sollte man erst prüfen, ob es dafür nicht schon einen passenden, fertigen Baustein in der Bibliothek gibt (hier: `GreenWhiteBackground3_AX`, siehe Uebung_010f3_AX). Der Vergleich beider SubApps im 4diac-Editor macht den Unterschied im Ressourcenverbrauch unmittelbar sichtbar.

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

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

2.1.2.155 Uebung_010f_AX: AuxFunction2_X1 auf DigitalOutput_Q1 mit GreenWhite-Background



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung erweitert `Uebung_010b1_AX`: Zusätzlich zur Ansteuerung von `DigitalOutput_Q1` wird der Zustand von `AuxFunction2_X1` als Hintergrundfarbe (grün = EIN, weiß = AUS) zurückgemeldet. Sie zeigt, wie ein einzelnes Aux-Adaptersignal über `AX_SPLIT_2` auf zwei Verbraucher verteilt wird und warum für ein Auxiliary-Function-Type-2-Objekt der spezielle Baustein `GreenWhiteBackground2_AX` benötigt wird.

Verwendete Funktionsbausteine (FBs)

- **AuxFunction2_X1**: Auxiliary-Eingang (Typ: `isobus::UT::io::Auxiliary::IN::Aux_IXA`)
  - **Parameter**: `QI = TRUE`, `u16ObjId = AuxFunction2_X1`
  - **Erklärung**: Liefert den booleschen Zustand des zugewiesenen physischen Aux-Tasters/Joysticks als AX-Adapterausgang `IN`.
- **AX_SPLIT_2**: Adapter-Signalverteiler (Typ: `adapter::events::unidirectional::AX_SPLIT_2`)
  - **Parameter**: keine
  - **Erklärung**: Vervielfältigt das eine AX-Signal von `AuxFunction2_X1` verlustfrei auf zwei Ziele, da ein Adapter-Plug nicht direkt an mehrere Sockets angeschlossen werden kann.
- **DigitalOutput_Q1**: logiBUS Digitalausgang (Typ: `logiBUS::io::DQ::logiBUS_QXA`)
  - **Parameter**: `QI = TRUE`, `Output = Output_Q1`
  - **Erklärung**: Schaltet den physischen Ausgang `Q1` entsprechend dem Aux-Signal.

Sub-Bausteine: GreenWhiteBackground2_AX

- **GreenWhiteBackground2_AX** (Typ: `MyLib::sys::GreenWhiteBackground2_AX`)
  - **Parameter**: `u16ObjIdA = AuxFunction2_X1`
  - **Erklärung**: Setzt die Hintergrundfarbe (grün/weiß) sowohl auf dem VT-Bildschirm als auch am Aux-Handle (Joystick) selbst. `GreenWhiteBackground2_AX` (nicht `GreenWhiteBackground1_AX`) ist hier zwingend nötig, weil `AuxFunction2_X1` ein Auxiliary-Function-Type-2-Objekt ist: seine Zuweisung wird an zwei getrennten Stellen angezeigt und muss daher intern sowohl `Q_BackgroundColour` (VT-Bildschirm) als auch `Q_BackgroundColourAux` (Aux-Handle) senden.

Programmablauf und Verbindungen

1. `AuxFunction2_X1.IN` liefert bei jeder Zustandsänderung des Aux-Tasters ein AX-Adaptersignal.
2. Dieses Signal wird über `AX_SPLIT_2.IN` empfangen und auf `OUT1`/`OUT2` dupliziert.
3. `AX_SPLIT_2.OUT1` ⟶ `DigitalOutput_Q1.OUT`: der physische Ausgang `Q1` folgt direkt dem Aux-Zustand.

4. `AX_SPLIT_2.OUT2` ⟶ `GreenWhiteBackground2_AX.DI1`: dieselbe Information steuert parallel die Hintergrundfarbe auf VT-Bildschirm und Aux-Handle.

Hinweis aus dem Netzwerk-Kommentar: `AuxFunction2_X1` muss zweimal angegeben werden (einmal an `Aux_IXA.u16ObjId`, einmal an `GreenWhiteBackground2_AX.u16ObjIdA`) – dieser Nachteil wird in `Uebung_010f2_AX` durch Kapselung in eine eigene Subapp behoben.

Zusammenfassung

Die Übung zeigt die Kombination aus Aux-Eingang, Adapter-Signalverteilung (`AX_SPLIT_2`) und der Auxiliary-Function-Type-2-spezifischen Hintergrundfarbrückmeldung (`GreenWhiteBackground2_AX`), die gleichzeitig VT-Bildschirm und Aux-Handle bedient. Sie bildet die Grundlage für die gekapselte Variante `Uebung_010f2_AX`.

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

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

2.1.2.156 Uebung_011_AUDI: Numeric Value Input

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_011_AUDI`. Sie ist die adapterbasierte Variante der Übung 011 und zeigt, wie numerische Werte effizient und übersichtlich verarbeitet werden.

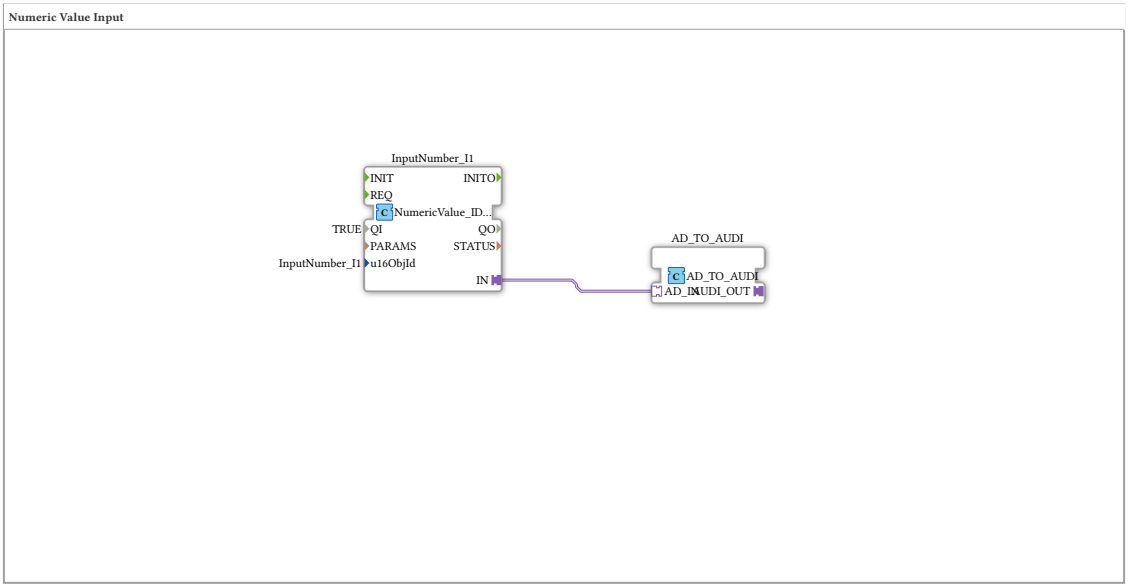
Ziel der Übung

Erlernen der modernen, adapterbasierten Verarbeitung von ISOBUS-Terminal-Eingaben. Durch die Verwendung von Adaptern wird das Baustein-Netzwerk kompakter und die Trennung von Ereignis- und Datenfluss erfolgt implizit innerhalb der Adapter-Struktur.

Beschreibung und Komponenten

Die Subapplikation `Uebung_011_AUDI.SUB` nutzt einen adapterbasierten Eingabe-Baustein.

Funktionsbausteine (FBs)



- **`InputNumber_I1`**: Typ `NumericValue_IDA`. Dieser Baustein stellt ein numerisches Eingabefeld auf dem ISOBUS-Terminal dar. Im Gegensatz zur Standard-Variante (`_ID`) nutzt dieser Baustein einen AX-basierten Adapter-Ausgang (`IN`), der sowohl das Ereignis als auch den DWORD-Wert bündelt.
- **`F_DWORD_TO_UDINT`**: Hier wird der neue Bausteintyp `AD_TO_AUDI` verwendet. Er nimmt den `AD`-Adapter entgegen und gibt einen `AUDI`-Adapter aus, der den Wert als `UDINT` führt.

Funktionsweise

Die Verbindung zwischen Eingabe und Konvertierung erfolgt ausschließlich über eine Adapter-Linie:

```
<AdapterConnections>
    <Connection Source="InputNumber_I1.IN" Destination="F_DWORD_TO_UDINT.AD_IN"/>
</AdapterConnections>
```

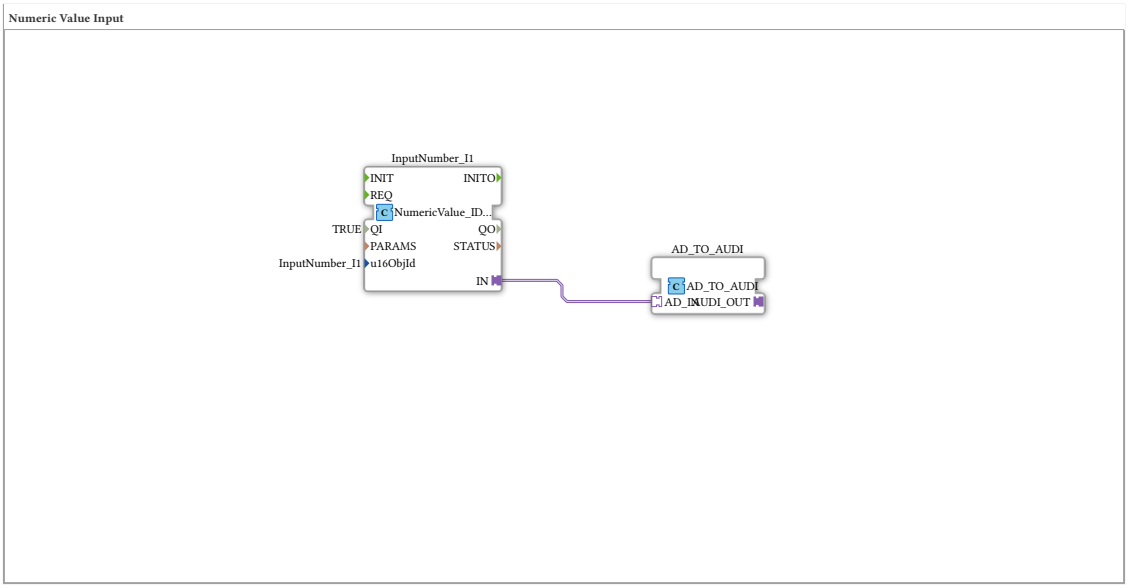
1. Der Nutzer gibt am Terminal einen Wert ein (z. B. „100“).
2. Nach der Bestätigung sendet der Baustein `InputNumber_I1` das Update über den Adapter-Plug.
3. Der Konverter `AD_TO_AUDI` (instanziiert als `F_DWORD_TO_UDINT`) empfängt dieses Paket, wandelt den Typ und stellt das Ergebnis am `AUDI`-Plug für die restliche Logik bereit.

Fazit

Die Übung verdeutlicht den Vorteil von Adaptern: Anstatt separate Linien für Ereignisse (`REQ`/`CNF`) und Daten (`IN`/`OUT`) ziehen zu müssen, reicht eine einzige Adapter-Verbindung aus. Dies reduziert die Fehleranfälligkeit und erhöht die Lesbarkeit des Programms erheblich.

2.1.2.157 Uebung_011_AX: Numeric Value Input

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_011_AX (Numeric Value Input).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Numeric Value Input**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_011_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **InputNumber_I1**: Instanz des Typs isobus::UT::io::NumericValue::NumericValue_IDA.
  - Parameter QI = TRUE
  - Parameter u16ObjId = InputNumber_I1
- **AD_TO_AUDI**: Instanz des Typs adapter::conversion::unidirectional::AD_TO_AUDI.

Verbindungen und Schnittstellen

Adapterverbindungen:

- InputNumber_I1.IN -> AD_TO_AUDI.AD_IN

Programmablauf und Funktionsweise

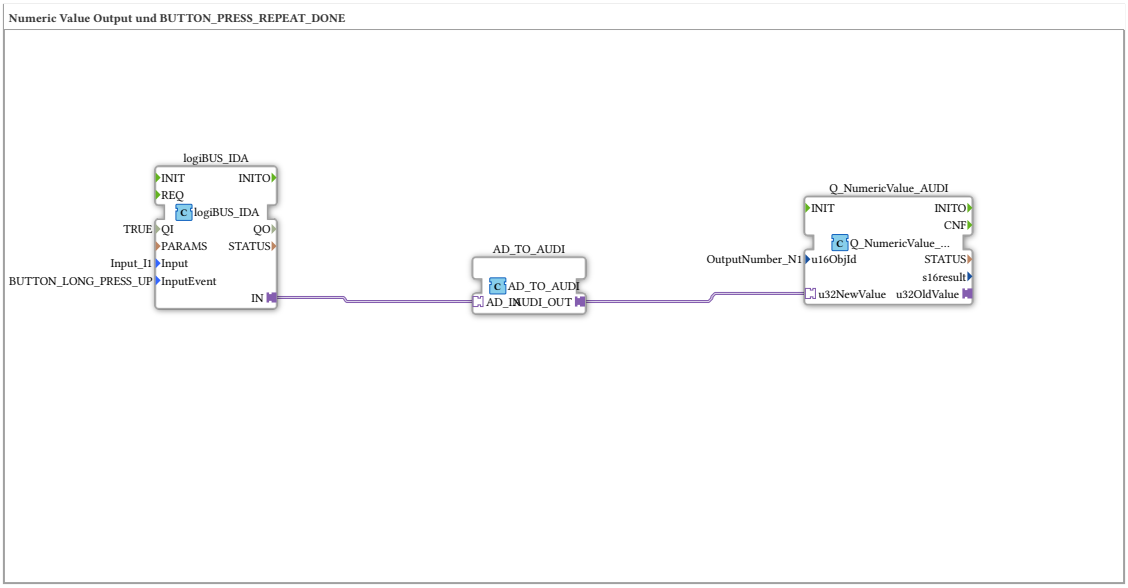
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.

2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_011_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.158 Uebung_011a2_AUDI: Numeric Value Output und BUTTON_PRESS_REPEAT_DONE



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung demonstriert die Ausgabe eines numerischen Werts unter Verwendung eines Tastendruck-Ereignisses (BUTTON_LONG_PRESS_UP). Dabei wird ein digitaler Eingangsbaustein (logiBUS_IDA) verwendet, der bei einem langen Tastendruck ein Ereignis auslöst. Dieses Ereignis wird über einen Adapter in ein Format konvertiert, das der Ausgabebaustein Q_NumericValue_AUDI verarbeiten kann – dieser gibt dann den vordefinierten numerischen Wert auf dem ISOBUS aus.

Verwendete Funktionsbausteine (FBs)

- **logiBUS_IDA**
  - **Typ**: logiBUS::io::DI::logiBUS_IDA
  - **Parameter**: QI = TRUE, Input = Input_I1, InputEvent = BUTTON_LONG_PRESS_UP
  - **Funktionsweise**: Stellt einen digitalen Eingang dar, der auf das Ereignis „langer Tastendruck“ (Long Press Up) reagiert. Bei Auslösung wird ein entsprechendes Signal an den Adapter weitergegeben.
- **AD_TO_AUDI**
  - **Typ**: adapter::conversion::unidirectional::AD_TO_AUDI
  - **Funktionsweise**: Dient als Adapter zur unidirektionalen Konvertierung von logiBUS-ADA-Daten in das Format, das der numerische Ausgabebaustein (Q_NumericValue_AUDI) erwartet. Er wandelt das Ereignis in einen Datenwert um.
- **Q_NumericValue_AUDI**
  - **Typ**: isobus::UT::Q::Q_NumericValue_AUDI
  - **Parameter**: u16ObjId = OutputNumber_N1

– **Funktionsweise**: Nimmt einen 32-Bit-Wert (hier über den Adapter) entgegen und gibt ihn über das ISOBUS-Objekt mit der Objekt-ID `OutputNumber_N1` aus. Dies ermöglicht die Anzeige eines numerischen Werts auf einem ISOBUS-Terminal.

Programmablauf und Verbindungen

Die Funktionsbausteine sind wie folgt miteinander verbunden:

1. **logiBUS_IDA -> AD_TO_AUDI (AD_IN)**:
   Bei einem langen Tastendruck (Ereignis `BUTTON_LONG_PRESS_UP`) erzeugt `logiBUS_IDA` ein Signal auf dem Ausgang `IN`, das an den Adaptereingang `AD_IN` weitergeleitet wird.
2. **AD_TO_AUDI (AUDI_OUT) -> Q_NumericValue_AUDI (u32NewValue)**:
   Der Adapter wandelt das eingehende Signal in einen numerischen Datenwert um und sendet ihn über den Ausgang `AUDI_OUT` an den Dateneingang `u32NewValue` des Ausgabebausteins.

Durch diese Kette wird bei jedem langen Tastendruck der festgelegte numerische Wert (hier das ISOBUS-Objekt `OutputNumber_N1`) ausgegeben.

Lernziele:

- Verwendung von digitalen Eingangsbausteinen mit Ereignisauslösung (Long Press).
- Einsatz von Adaptern zur Konvertierung zwischen verschiedenen Protokoll-/Datenformaten.
- Ausgabe numerischer Werte über ISOBUS-Objekte.

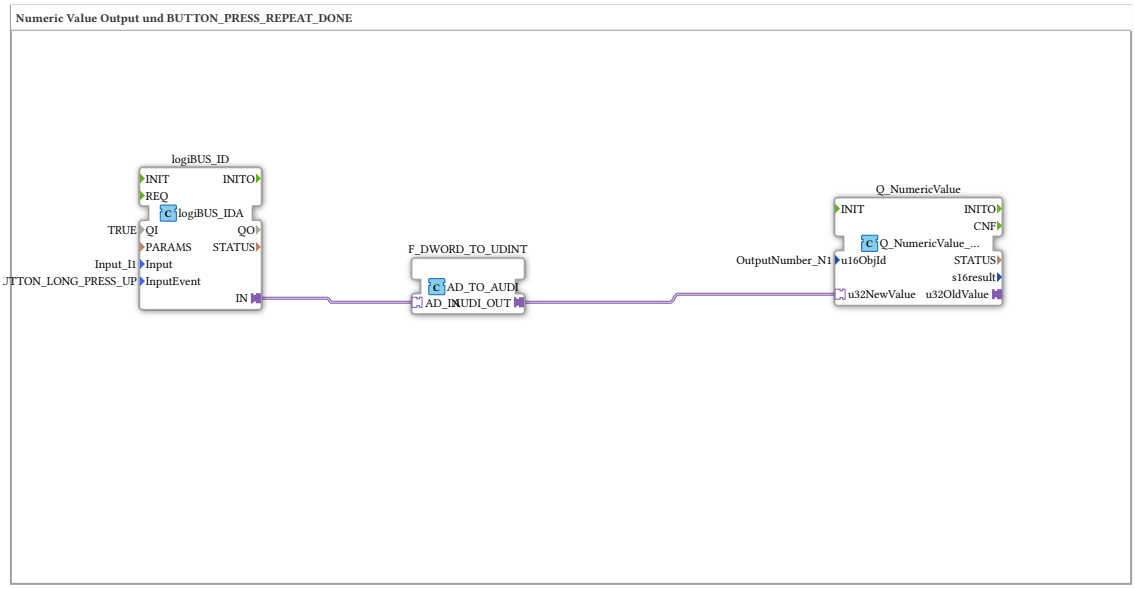
Schwierigkeitsgrad: Einsteiger / Grundlagen
Benötigte Vorkenntnisse: Grundlegendes Verständnis von Funktionsbausteinen, Ereignissen und ISOBUS-Kommunikation.

Zusammenfassung

Die Übung **Uebung_011a2_AUDI** zeigt einen kompakten Ablauf: Ein digitaler Tastendruck (lang) löst eine Ereigniskette aus, an deren Ende ein numerischer Wert auf dem ISOBUS ausgegeben wird. Die drei verwendeten Bausteine – der Eingangsbaustein, ein Konvertierungsadapter und der Ausgabebaustein – sind über Adapterverbindungen lose gekoppelt. Dies ermöglicht eine flexible, ereignisgesteuerte Werteausgabe und vermittelt grundlegende Konzepte der modularen Steuerungsprogrammierung.

2.1.2.159 Uebung_011a2_AX: Numeric Value Output und BUTTON_PRESS_REPEAT_DONE

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_011a2_AX (Numeric Value Output und BUTTON_PRESS_REPEAT_DONE).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Numeric Value Output und BUTTON_PRESS_REPEAT_DONE**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_011a2_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **Q_NumericValue**: Instanz des Typs isobus::UT::Q::Q_NumericValue_AUDI.
  - Parameter u16ObjId = OutputNumber_N1
- **logiBUS_ID**: Instanz des Typs logiBUS::io::DI::logiBUS_IDA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
  - Parameter InputEvent = BUTTON_LONG_PRESS_UP
- **F_DWORD_TO_UDINT**: Instanz des Typs adapter::conversion::unidirectional::AD_TO_AUDI.

Verbindungen und Schnittstellen

Adapterverbindungen:

- logiBUS_ID.IN -> F_DWORD_TO_UDINT.AD_IN
- F_DWORD_TO_UDINT.AUDI_OUT -> Q_NumericValue.u32NewValue

Hinweise aus dem Modell

BUTTON_PRESS_UP BUTTON_LONG_PRESS_HOLD BUTTON_LONG_PRESS_UP

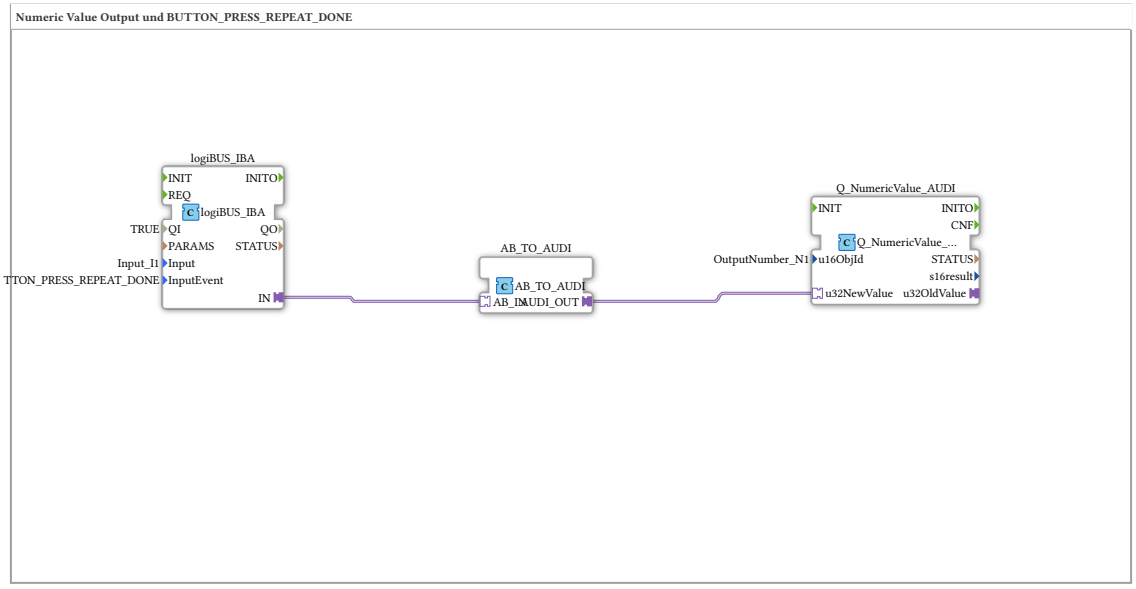
Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_011a2_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.160 Uebung_011a_AUDI: Numeric Value Output und BUTTON_PRESS_REPEAT_DONE



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung demonstriert die Verwendung eines digitalen Eingangs (Taster) mit Wiederholerkennung (`BUTTON_PRESS_REPEAT_DONE`) zur Ausgabe eines numerischen Werts auf einem ISOBUS Virtual Terminal. Der eingehende Tastendruck wird über eine logiBUS-IBA-Schnittstelle erfasst, über einen Adapter in ein ISOBUS-kompatibles Format umgewandelt und schließlich an einen `Q_NumericValue_AUDI`-Baustein gesendet, der den Wert auf dem Display des Terminals darstellt.

Die Übung basiert auf dem Standard IEC 61499 und nutzt die vordefinierten Bibliotheken `logiBUS` und `isobus`. Sie eignet sich für Einsteiger in die 4diac-IDE, die sich mit der Verknüpfung von physikalischen Eingängen und ISOBUS-VT-Komponenten vertraut machen möchten.

Verwendete Funktionsbausteine (FBs)

Im SubApp `Uebung_011a_AUDI` kommen drei Funktionsbausteine zum Einsatz:

- **logiBUS_IBA**
  - **Typ**: `logiBUS::io::DI::logiBUS_IBA`
  - **Parameter**:
    - `QI` = `TRUE` (Initialisierung aktiv)
    - `Input` = `Input_I1` (physikalischer Digitaleingang)
    - `InputEvent` = `BUTTON_PRESS_REPEAT_DONE` (Ereignis bei Tastendruck mit Wiederholung)
  - **Funktion**: Dieser Baustein liest einen digitalen Eingang (Taster) aus. Das Ereignis `BUTTON_PRESS_REPEAT_DONE` wird ausgelöst, sobald der Taster gedrückt wird – inklusive Wiederholfunktion (z. B. für langes Drücken). Der gelesene Wert wird über den Adapterausgang `IN` als AB-Format (Arbeitsblatt) bereitgestellt.
- **AB_TO_AUDI**
  - **Typ**: `adapter::conversion::unidirectional::AB_TO_AUDI`
  - **Parameter**: keine (reine Konvertierung)
  - **Funktion**: Dieser Adapter wandelt das AB-Format (interne Darstellung der logiBUS-Eingänge) in das AUDI-Format um, das von den `isobus`-Bausteinen verarbeitet werden kann. Die Umwandlung erfolgt unidirektional.
- **Q_NumericValue_AUDI**
  - **Typ**: `isobus::UT::Q::Q_NumericValue_AUDI`
  - **Parameter**:
    - `u16ObjId` = `OutputNumber_N1` (ID des numerischen Ausgabeobjekts auf dem Virtual Terminal)

– **Funktion**: Dieser Baustein empfängt einen numerischen Wert im AUDI-Format (über `u32NewValue`) und sendet diesen an das auf dem ISOBUS-VT definierte Objekt mit der ID `OutputNumber_N1`. Der Wert wird dort sichtbar dargestellt.

Hinweis: Die Übung enthält keine eigenen Sub-Bausteine (SubAppTypes). Die genannten FBs stammen aus den importierten Bibliotheken `logiBUS` und `isobus`.

Programmablauf und Verbindungen

1. **Ereignisauslösung** Ein Druck auf den Taster (definiert als `Input_I1`) mit Wiederholerkennung erzeugt das Ereignis `BUTTON_PRESS_REPEAT_DONE`. Dieses Ereignis aktiviert den Baustein `logiBUS_IBA`.
2. **Einlesen des Eingangs** `logiBUS_IBA` liest den aktuellen Zustand des Digitaleingangs und stellt ihn über den Adapterausgang `IN` als AB-Format bereit.
3. **Formatkonvertierung** Der Adapter `AB_TO_AUDI` wandelt das AB-Format in das AUDI-Format um. Die Verbindung erfolgt über eine Adapterverbindung (`AdapterConnections`):
   - `Source="logiBUS_IBA.IN"` ⟶ `Destination="AB_TO_AUDI.AB_IN"`
4. **Ausgabe auf dem Virtual Terminal** Der konvertierte Wert (AUDI-Format) wird am Ausgang `AUDI_OUT` von `AB_TO_AUDI` bereitgestellt und über eine weitere Adapterverbindung an den Baustein `Q_NumericValue_AUDI` übergeben:
   - `Source="AB_TO_AUDI.AUDI_OUT"` ⟶ `Destination="Q_NumericValue_AUDI.u32NewValue"`

   `Q_NumericValue_AUDI` aktualisiert daraufhin das auf dem ISOBUS-VT hinterlegte numerische Objekt mit der ID `OutputNumber_N1`.

Zusammenfassender Datenfluss:
`Taster (Input_I1) -> BUTTON_PRESS_REPEAT_DONE -> logiBUS_IBA -> AB_TO_AUDI -> Q_NumericValue_AUDI -> VT-Objekt OutputNumber_N1`
Die gesamte Logik ist in einem SubApp `Uebung_011a_AUDI` gekapselt, der keinerlei eigene Ein-/Ausgangsschnittstellen besitzt (leeres `SubAppInterfaceList`). Die Verbindungen nach außen erfolgen ausschließlich über die verwendeten Bausteine, deren Parameter auf globale Konstanten (`Input_I1`, `OutputNumber_N1`, `BUTTON_PRESS_REPEAT_DONE`) verweisen.

Zusammenfassung

In dieser Übung wird die grundlegende Interaktion zwischen physikalischen Tastern und einem ISOBUS Virtual Terminal realisiert. Die Teilnehmer lernen:

- Wie ein digitaler Eingang mit Wiederholerkennung (`BUTTON_PRESS_REPEAT_DONE`) in 4diac konfiguriert wird.
- Wie Adapter zur Formatkonvertierung (AB ↔ AUDI) eingesetzt werden.
- Wie ein numerischer Wert über `Q_NumericValue_AUDI` auf dem ISOBUS-VT ausgegeben wird.

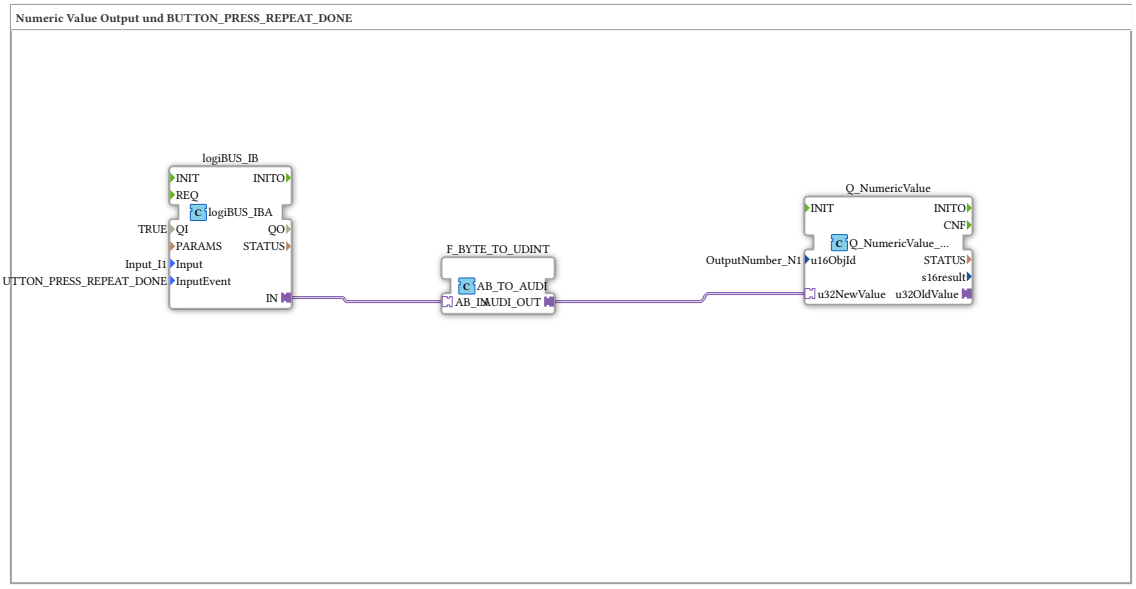
Die Übung eignet sich als Einstieg in die ISOBUS-Kommunikation mit der 4diac-IDE. Vorausgesetzt werden grundlegende Kenntnisse der IEC 61499 sowie die Installation der Bibliotheken `logiBUS` und `isobus`. Der SubApp kann direkt in ein Anwendungsprojekt eingebunden und getestet werden.

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

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

2.1.2.161 Uebung_011a_AX: Numeric Value Output und BUTTON_PRESS_REPEAT_DONE

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_011a_AX (Numeric Value Output und BUTTON_PRESS_REPEAT_DONE).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Numeric Value Output und BUTTON_PRESS_REPEAT_DONE**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_011a_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **Q_NumericValue**: Instanz des Typs isobus::UT::Q::Q_NumericValue_AUDI.
  - Parameter u16ObjId = OutputNumber_N1
- **logiBUS_IB**: Instanz des Typs logiBUS::io::DI::logiBUS_IBA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
  - Parameter InputEvent = BUTTON_PRESS_REPEAT_DONE
- **F_BYTE_TO_UDINT**: Instanz des Typs adapter::conversion::unidirectional::AB_TO_AUDI.

Verbindungen und Schnittstellen

Adapterverbindungen:

- logiBUS_IB.IN -> F_BYTE_TO_UDINT.AB_IN
- F_BYTE_TO_UDINT.AUDI_OUT -> Q_NumericValue.u32NewValue

Hinweise aus dem Modell

> BUTTON_PRESS_REPEAT_DONE: nur am Ende 1 Event BUTTON_PRESS_REPEAT: Während des drückens kommen Events

---

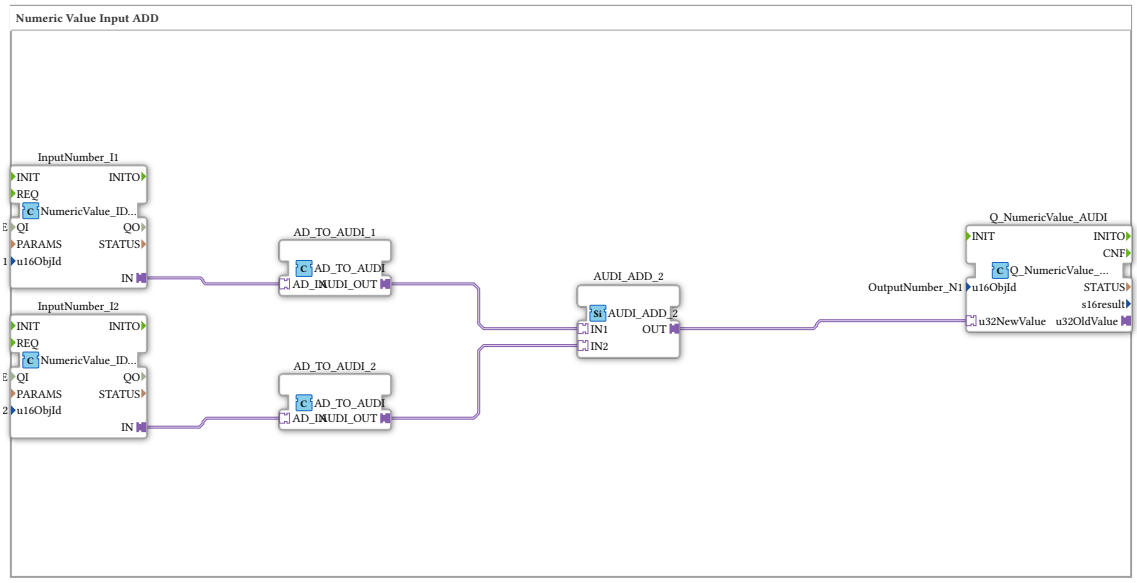
Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_011a_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.162 Uebung_011b1_AUDI: Numeric Value Input ADD



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert die Verarbeitung von zwei numerischen ISOBUS-Eingangswerten und deren Addition mithilfe spezieller Adapterbausteine. Die beiden eingehenden Werte (ganze Zahlen) werden über einen `AD_TO_AUDI`-Adapter in einen „AUDI“-kompatiblen Typ konvertiert, anschließend in einem Additionsbaustein (`AUDI_ADD_2`) aufsummiert und das Ergebnis über einen Ausgabebaustein (`Q_NumericValue_AUDI`) als ISOBUS-Ausgangsobjekt bereitgestellt.

Der Fokus liegt auf dem Umgang mit Adapterverbindungen und der Typkonvertierung zwischen unterschiedlichen Datenformaten in 4diac.

Verwendete Funktionsbausteine (FBs)

Die SubApp `Uebung_011b1_AUDI` enthält sechs Funktionsbausteine, die über Adapter verbunden sind.

Sub-Bausteine: `InputNumber_I1` und `InputNumber_I2`

- **Typ**: `isobus::UT::io::NumericValue::NumericValue_IDA`
- **Parameter**:
  - `QI` = `TRUE`
  - `u16ObjId` = `InputNumber_I1` bzw. `InputNumber_I2`
- **Funktionsweise**: Diese Bausteine stellen den Zugriff auf zwei ISOBUS-NumericValue-Eingangsobjekte dar. Sie liefern an ihrem Ausgangsadapter `IN` einen numerischen Wert (ganze Zahl) vom ISOBUS-Datentyp. Der Parameter `u16ObjId` definiert die Kennung des jeweiligen Eingangsobjekts.

Sub-Bausteine: `AD_TO_AUDI_1` und `AD_TO_AUDI_2`

- **Typ**: `adapter::conversion::unidirectional::AD_TO_AUDI`
- **Parameter**: keine
- **Funktionsweise**: Diese Bausteine konvertieren einen Eingangswert vom Adaptertyp `AD` (der von den vorherigen NumericValue-Bausteinen bereitgestellt wird) in den Adaptertyp `AUDI`. Die Umwandlung erfolgt unidirektional und dient der Anpassung an den nachfolgenden Additionsbaustein, der nur den `AUDI`-Typ akzeptiert.

Sub-Baustein: `AX_ADD_2`

- **Typ**: `adapter::iec61131::arithmetic::AUDI_ADD_2`
- **Parameter**: keine

- **Funktionsweise**: Dieser Baustein führt eine Addition zweier `AUDI`-Eingangswerte durch. Die Eingänge `IN1` und `IN2` werden über Adapterverbindungen mit den konvertierten Werten versorgt. Der Ausgang `OUT` liefert das Ergebnis als `AUDI`-Summe.

Sub-Baustein: `Q_NumericValue_AUDI`

- **Typ**: `isobus::UT::Q::Q_NumericValue_AUDI`
- **Parameter**:
  - `u16ObjId` = `OutputNumber_N1`
- **Funktionsweise**: Dieser Baustein nimmt über den Adaptereingang `u32NewValue` den berechneten Summenwert (als `AUDI`-Typ) entgegen und schreibt ihn als ISOBUS-Ausgangsobjekt mit der Kennung `OutputNumber_N1`. Er fungiert somit als Ausgabe-Schnittstelle zur angeschlossenen ISOBUS-Steuerung.

Programmablauf und Verbindungen

1. Die beiden ISOBUS-Eingänge `InputNumber_I1` und `InputNumber_I2` liefern über ihre `IN`-Adapterausgänge numerische Werte.
2. Jeweils ein `AD_TO_AUDI`-Baustein (`AD_TO_AUDI_1` und `AD_TO_AUDI_2`) konvertiert diese Werte in den `AUDI`-Adaptertyp.
3. Die konvertierten Werte werden an die Eingänge `IN1` und `IN2` des Additionsbausteins `AX_ADD_2` übergeben.
4. `AX_ADD_2` berechnet die Summe und stellt sie an `OUT` bereit.
5. Der Summenwert gelangt an den `u32NewValue`-Eingang des Ausgabebausteins `Q_NumericValue_AUDI`, der ihn schließlich als ISOBUS-Ausgangsobjekt `OutputNumber_N1` ausgibt.

Die gesamte Datenverarbeitung erfolgt in einem Zyklus ohne zusätzliche Ereignissteuerung – die Bausteine werden automatisch durchlaufen, sobald die Eingangswerte anliegen.

Zusammenfassung

Die Übung zeigt eine vollständige Verarbeitungskette für zwei numerische ISOBUS-Eingangswerte:

- Einlesen über standardisierte ISOBUS-Bausteine (`NumericValue_IDA`),
- Typkonvertierung mittels Adapter-Bausteinen (`AD_TO_AUDI`),
- arithmetische Addition (`AUDI_ADD_2`),
- und Ausgabe über einen ISOBUS-Ausgangsbaustein (`Q_NumericValue_AUDI`).

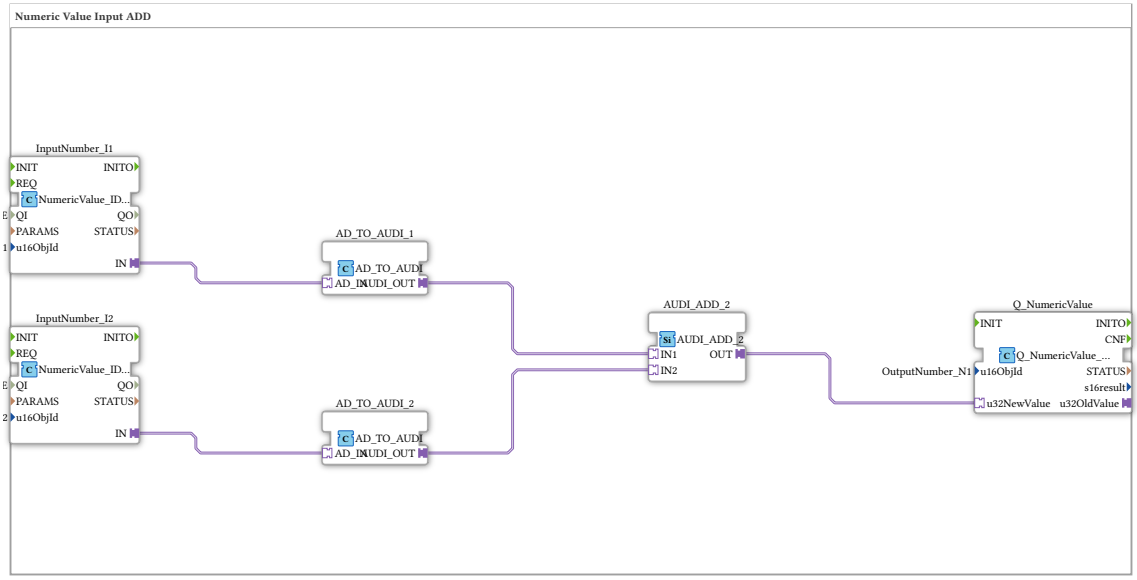
Sie vermittelt grundlegende Kenntnisse im Umgang mit Adapterschnittstellen und der Datenflussprogrammierung in 4diac für industrielle ISOBUS-Anwendungen (z. B. Landmaschinensteuerungen).

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

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

2.1.2.163 Uebung_011b1_AX: Numeric Value Input ADD

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_011b1_AX (Numeric Value Input ADD).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Numeric Value Input ADD**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_011b1_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **InputNumber_I1**: Instanz des Typs isobus::UT::io::NumericValue::NumericValue_IDA.
  - Parameter QI = TRUE
  - Parameter u16ObjId = InputNumber_I1
- **InputNumber_I2**: Instanz des Typs isobus::UT::io::NumericValue::NumericValue_IDA.
  - Parameter QI = TRUE
  - Parameter u16ObjId = InputNumber_I2
- **AD_TO_AUDI_1**: Instanz des Typs adapter::conversion::unidirectional::AD_TO_AUDI.
- **AD_TO_AUDI_2**: Instanz des Typs adapter::conversion::unidirectional::AD_TO_AUDI.
- **AUDI_ADD_2**: Instanz des Typs adapter::iec61131::arithmetic::AUDI_ADD_2.
- **Q_NumericValue**: Instanz des Typs isobus::UT::Q::Q_NumericValue_AUDI.
  - Parameter u16ObjId = OutputNumber_N1

Verbindungen und Schnittstellen

Adapterverbindungen:

- InputNumber_I1.IN -> AD_TO_AUDI_1.AD_IN
- InputNumber_I2.IN -> AD_TO_AUDI_2.AD_IN
- AD_TO_AUDI_1.AUDI_OUT -> AUDI_ADD_2.IN1
- AD_TO_AUDI_2.AUDI_OUT -> AUDI_ADD_2.IN2
- AUDI_ADD_2.OUT -> Q_NumericValue.u32NewValue

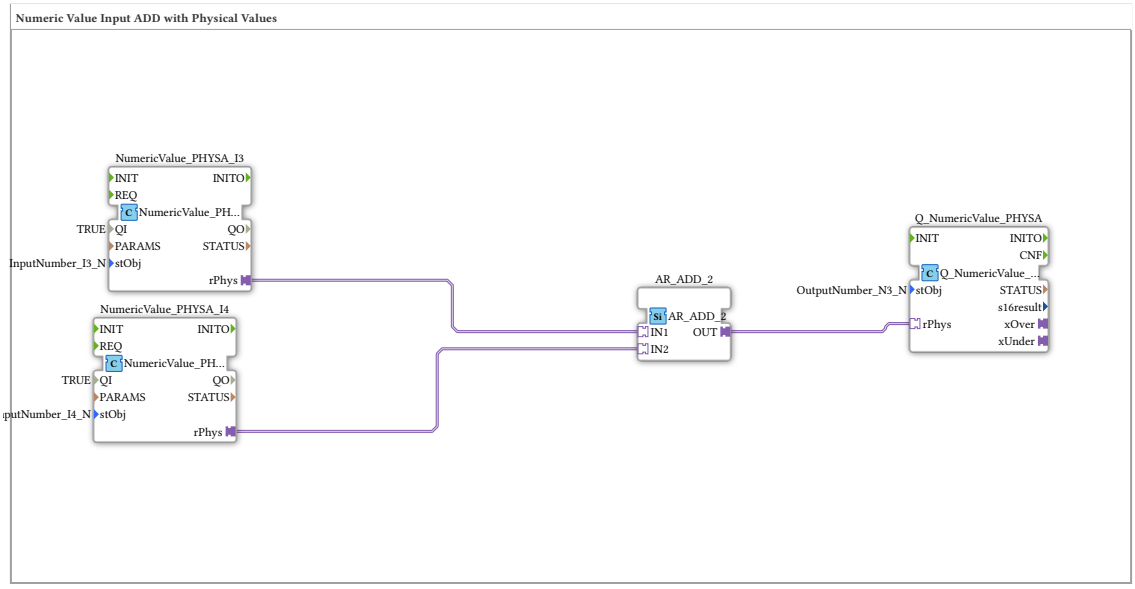
Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_011b1_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.164 Uebung_011b1_PHYSA: Numeric Value Input ADD with Physical Values



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert die Verarbeitung von physikalischen Messwerten (z.B. Spannung, Strom, Drehzahl) durch eine arithmetische Operation. Zwei Eingangswerte aus definierten physikalischen Quellen werden mittels eines Additionsbausteins verknüpft und das Ergebnis an einen physikalischen Ausgang weitergegeben. Der Fokus liegt auf der korrekten Verdrahtung der Adapter-Schnittstellen zwischen den Funktionsbausteinen (FBs) zur Signalkopplung mit realen I/O-Kanälen.

Verwendete Funktionsbausteine (FBs)

Die Übung besteht aus vier direkt instanziierten Funktionsbausteinen. Es sind keine weiteren Sub-Bausteine (SubApp) enthalten.

Name	Typ	Beschreibung
`NumericValue_PHYSA_I3`	`isobus::UT::io::NumericValue::NumericValue_PHYSA`	Liest den physikalischen Wert von der Hardwareschnittstelle `InputNumber_I3` und stellt ihn als physikalische Größe (rPhys) bereit.
`NumericValue_PHYSA_I4`	`isobus::UT::io::NumericValue::NumericValue_PHYSA`	Gleiche Funktion wie oben, jedoch für die Schnittstelle `InputNumber_I4`.
`AR_ADD_2`	`adapter::iec61131::arithmetic::AR_ADD_2`	Führt eine Addition zweier physikalischer Werte durch (IN1 + IN2) und gibt das Ergebnis als OUT aus.
`Q_NumericValue_PHYSA`	`isobus::UT::Q::Q_NumericValue_PHYSA`	Schreibt den übergebenen physikalischen Wert auf die Hardwareschnittstelle `OutputNumber_N3`.

Parameter der einzelnen Instanzen

NumericValue_PHYSA_I3

- `QI` = TRUE (Aktivierung)

- `stObj` = `InputNumber_I3` (Objektname der Hardwareschnittstelle)

NumericValue_PHYSA_I4

- `QI` = TRUE
- `stObj` = `InputNumber_I4`

Q_NumericValue_PHYSA

- `stObj` = `OutputNumber_N3` (Objektname der Ausgangsschnittstelle)

AR_ADD_2 – keine Parameter gesetzt, alle Werte werden über Adapterverbindungen übergeben.

Programmablauf und Verbindungen

Das Netzwerk verbindet die Bausteine ausschließlich über Adapter-Kanäle (Typ `AdapterConnections`). Der Datenfluss ist linear:

1. **Eingangsmanagement**: Die Funktionsbausteine `NumericValue_PHYSA_I3` und `NumericValue_PHYSA_I4` lesen kontinuierlich die physikalischen Istwerte ihrer jeweiligen Hardware-Kanäle (`InputNumber_I3`, `InputNumber_I4`) und stellen diese an ihrem Adapter-Ausgang `rPhys` bereit.
2. **Arithmetische Verknüpfung**: Der Baustein `AR_ADD_2` empfängt die beiden physikalischen Werte über die Adaptereingänge `IN1` (von `NumericValue_PHYSA_I3.rPhys`) und `IN2` (von `NumericValue_PHYSA_I4.rPhys`) und addiert sie. Das Ergebnis wird am Adapter-Ausgang `OUT` ausgegeben.
3. **Ausgabe**: Der Adapter-Ausgang `AR_ADD_2.OUT` ist mit dem Adapter-Eingang `Q_NumericValue_PHYSA.rPhys` verbunden. Der Ausgangsbaustein übernimmt diesen Wert und schreibt ihn auf die Hardware-Schnittstelle `OutputNumber_N3`.

Die Verbindungen im Detail:

- `NumericValue_PHYSA_I3.rPhys` ⟶ `AR_ADD_2.IN1`
- `NumericValue_PHYSA_I4.rPhys` ⟶ `AR_ADD_2.IN2`
- `AR_ADD_2.OUT` ⟶ `Q_NumericValue_PHYSA.rPhys`

Zusammenfassung

Die Übung **Uebung_011b1_PHYSA** veranschaulicht den Aufbau einer typischen Messwertverarbeitungskette in der 4diac-IDE unter Einbeziehung physikalischer I/O-Kanäle. Sie zeigt, wie zwei analoge Eingangswerte (z.B. Spannungen) aus unterschiedlichen Kanälen eingelesen, addiert und auf einen analogen Ausgang geschrieben werden. Die komplette Signalverkettung erfolgt ausschließlich über Adapterverbindungen – ein wesentliches Konzept für modulare und hardwareunabhängige Automatisierungslösungen.

Lernziele:

- Verständnis der Adapter-Schnittstellen `rPhys` und `stObj`
- Zusammenspiel von Eingabe-, Rechen- und Ausgabebausteinen über Adapterverbindungen
- Umgang mit parametrierbaren I/O-Objekten (`InputNumber_I3`, `OutputNumber_N3`)
- Grundlagen der physikalischen Wertverarbeitung im 4diac-Umfeld

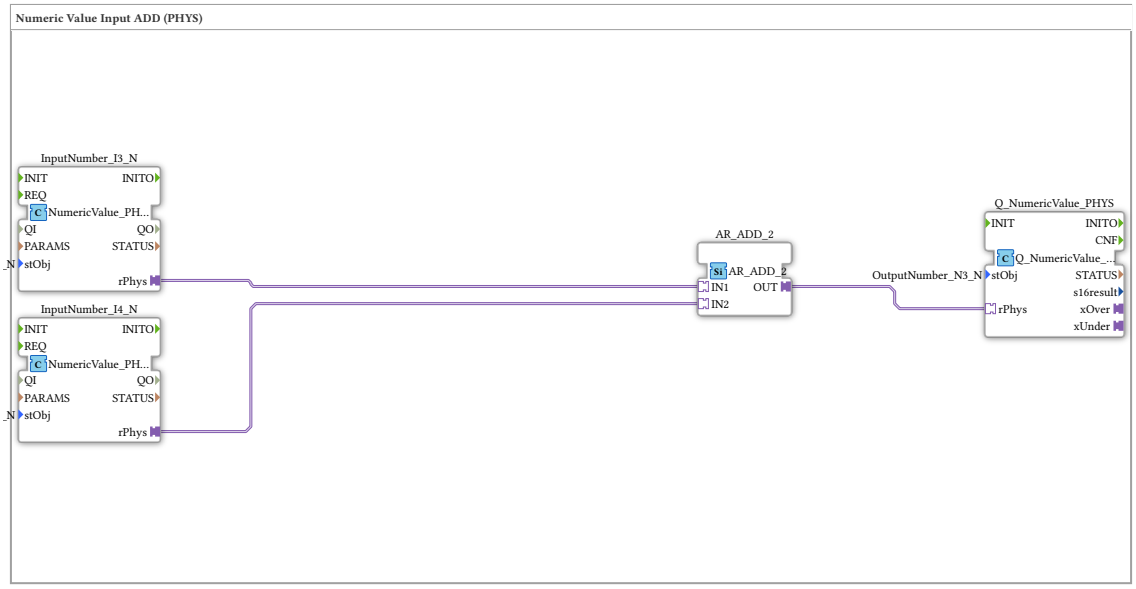
---

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

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

2.1.2.165 Uebung_011b1_PHYS_AX: Numeric Value Input ADD (PHYS)

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_011b1_PHYS_AX (Numeric Value Input ADD (PHYS)).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Numeric Value Input ADD (PHYS)**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_011b1_PHYS_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **InputNumber_I3_N**: Instanz des Typs isobus::UT::io::NumericValue::NumericValue_PHYSA.
  - Parameter stObj = InputNumber_I3_N
- **InputNumber_I4_N**: Instanz des Typs isobus::UT::io::NumericValue::NumericValue_PHYSA.
  - Parameter stObj = InputNumber_I4_N
- **AR_ADD_2**: Instanz des Typs adapter::iec61131::arithmetic::AR_ADD_2.
- **Q_NumericValue_PHYS**: Instanz des Typs isobus::UT::Q::Q_NumericValue_PHYSA.
  - Parameter stObj = OutputNumber_N3_N

Verbindungen und Schnittstellen

Adapterverbindungen:

- InputNumber_I3_N.rPhys -> AR_ADD_2.IN1
- InputNumber_I4_N.rPhys -> AR_ADD_2.IN2
- AR_ADD_2.OUT -> Q_NumericValue_PHYS.rPhys

Programmablauf und Funktionsweise

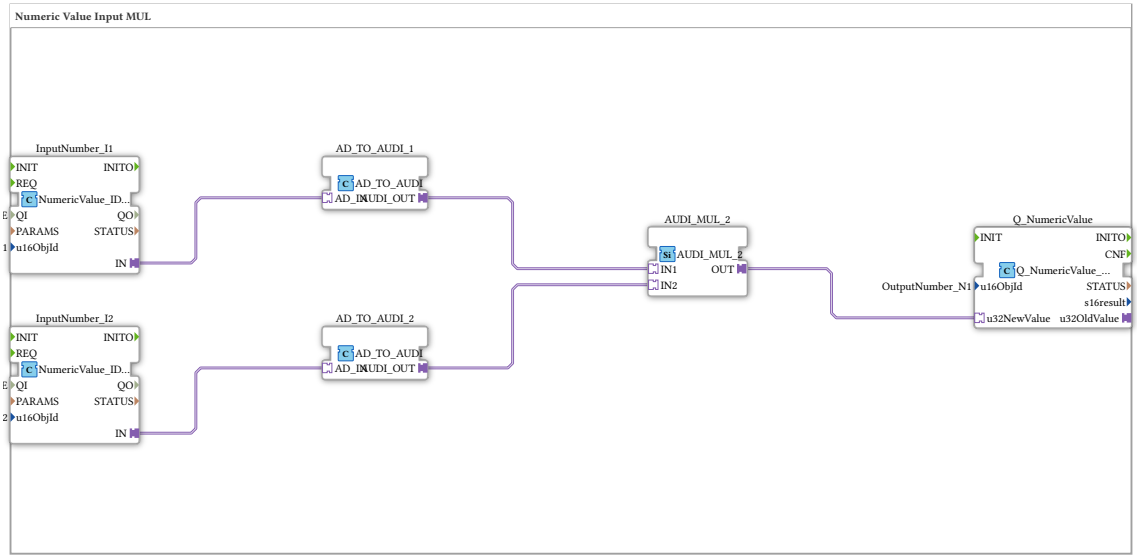
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_011b1_PHYS_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.166 Uebung_011b2_AX: Numeric Value Input MUL

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_011b2_AX (Numeric Value Input MUL).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Numeric Value Input MUL**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_011b2_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **InputNumber_I1**: Instanz des Typs isobus::UT::io::NumericValue::NumericValue_IDA.
  - Parameter QI = TRUE
  - Parameter u16ObjId = InputNumber_I1
- **InputNumber_I2**: Instanz des Typs isobus::UT::io::NumericValue::NumericValue_IDA.
  - Parameter QI = TRUE
  - Parameter u16ObjId = InputNumber_I2
- **AD_TO_AUDI_1**: Instanz des Typs adapter::conversion::unidirectional::AD_TO_AUDI.
- **AD_TO_AUDI_2**: Instanz des Typs adapter::conversion::unidirectional::AD_TO_AUDI.
- **AUDI_MUL_2**: Instanz des Typs adapter::iec61131::arithmetic::AUDI_MUL_2.
- **Q_NumericValue**: Instanz des Typs isobus::UT::Q::Q_NumericValue_AUDI.
  - Parameter u16ObjId = OutputNumber_N1

Verbindungen und Schnittstellen

Adapterverbindungen:

- InputNumber_I1.IN -> AD_TO_AUDI_1.AD_IN
- InputNumber_I2.IN -> AD_TO_AUDI_2.AD_IN
- AD_TO_AUDI_1.AUDI_OUT -> AUDI_MUL_2.IN1
- AD_TO_AUDI_2.AUDI_OUT -> AUDI_MUL_2.IN2
- AUDI_MUL_2.OUT -> Q_NumericValue.u32NewValue

Programmablauf und Funktionsweise

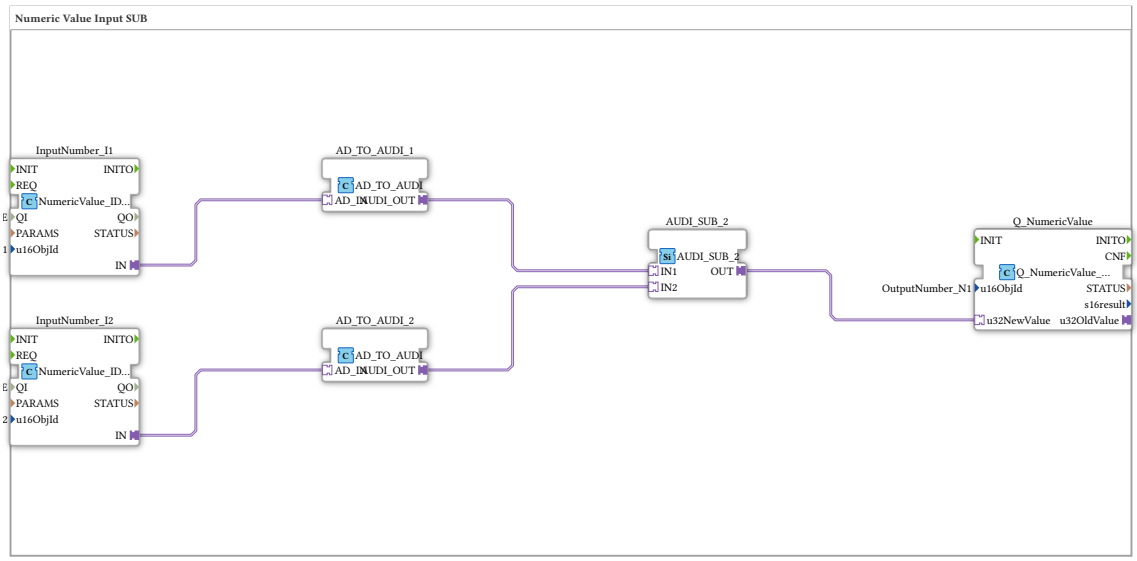
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_011b2_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.167 Uebung_011b3_AX: Numeric Value Input SUB

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_011b3_AX (Numeric Value Input SUB).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Numeric Value Input SUB**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_011b3_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **InputNumber_I1**: Instanz des Typs isobus::UT::io::NumericValue::NumericValue_IDA.
  - Parameter QI = TRUE
  - Parameter u16ObjId = InputNumber_I1
- **InputNumber_I2**: Instanz des Typs isobus::UT::io::NumericValue::NumericValue_IDA.
  - Parameter QI = TRUE
  - Parameter u16ObjId = InputNumber_I2
- **AD_TO_AUDI_1**: Instanz des Typs adapter::conversion::unidirectional::AD_TO_AUDI.
- **AD_TO_AUDI_2**: Instanz des Typs adapter::conversion::unidirectional::AD_TO_AUDI.
- **AUDI_SUB_2**: Instanz des Typs adapter::iec61131::arithmetic::AUDI_SUB_2.
- **Q_NumericValue**: Instanz des Typs isobus::UT::Q::Q_NumericValue_AUDI.
  - Parameter u16ObjId = OutputNumber_N1

Verbindungen und Schnittstellen

Adapterverbindungen:

- InputNumber_I1.IN -> AD_TO_AUDI_1.AD_IN
- InputNumber_I2.IN -> AD_TO_AUDI_2.AD_IN
- AD_TO_AUDI_1.AUDI_OUT -> AUDI_SUB_2.IN1
- AD_TO_AUDI_2.AUDI_OUT -> AUDI_SUB_2.IN2
- AUDI_SUB_2.OUT -> Q_NumericValue.u32NewValue

Programmablauf und Funktionsweise

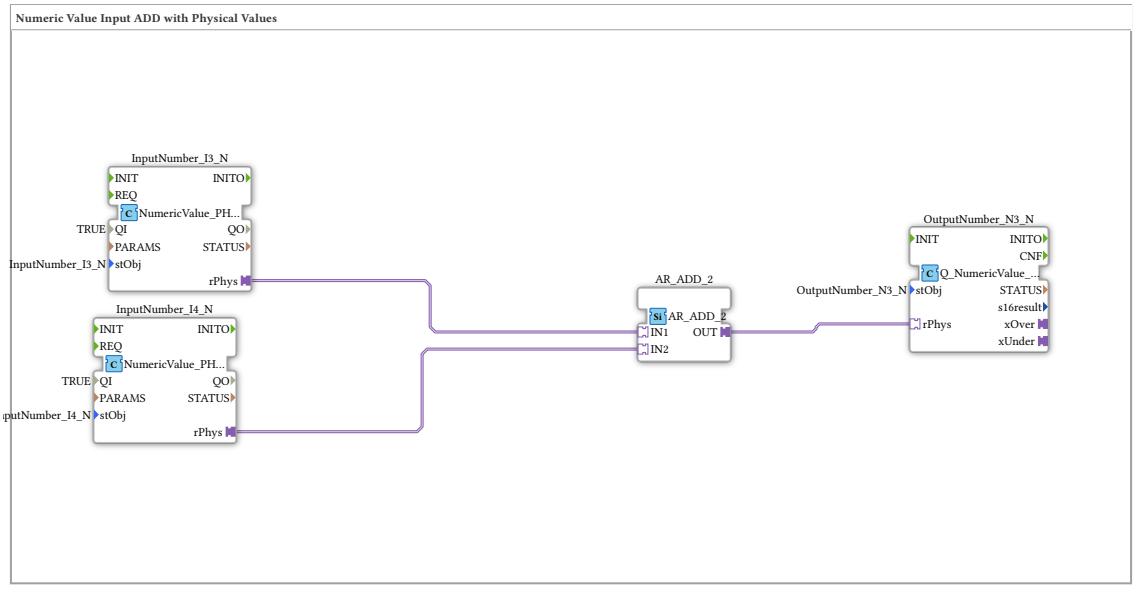
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.

2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_011b3_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.168 Uebung_011b3_PHYSA: Numeric Value Input ADD with Physical Values



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert die Verwendung von physikalischen Werten (Physical Values) in einer einfachen Additionsschaltung. Zwei numerische Eingänge liefern physikalische Größen, die mittels eines Additionsbausteins verrechnet werden. Das Ergebnis wird als physikalischer Wert ausgegeben. Ziel ist es, den Umgang mit Adapterverbindungen zwischen **NumericValue_PHYSA**-Bausteinen und dem standardisierten **AR_ADD_2**-Baustein zu erlernen.

Verwendete Funktionsbausteine (FBs)

Bausteinname	Typ	Kurzbeschreibung
InputNumber_I3	`isobus::UT::io::NumericValue::NumericValue_PHYSA`	Physikalische Werteingabe (z. B. mit Einheit). Parameter: `QI` = TRUE, `stObj` = „InputNumber_I3“. Stellt den eingegebenen physikalischen Wert über den Adapterausgang `rPhys` bereit.
InputNumber_I4	`isobus::UT::io::NumericValue::NumericValue_PHYSA`	Gleicher Typ wie InputNumber_I3. Parameter: `QI` = TRUE, `stObj` = „InputNumber_I4“.
AR_ADD_2	`adapter::iec61131::arithmetic::AR_ADD_2`	Addierer aus der IEC-61131-Arithmetikbibliothek. Nimmt zwei physikalische Werte an den Adaptereingängen `IN1` und `IN2` entgegen und liefert die Summe am Adapterausgang `OUT`.

OutputNumber_N3	`isobus::UT::Q::Q_NumericValue_PHYSA`	Physikalische Wertausgabe. Nimmt einen physikalischen Wert über den Adaptereingang `rPhys` entgegen und stellt ihn als Ausgabewert zur Verfügung. Parameter: `stObj` = „OutputNumber_N3“.

Hinweis: Es werden keine Sub-Bausteine (SubAppTypes) innerhalb dieser Übung verwendet.

Programmablauf und Verbindungen

Der Ablauf ist vollständig über Adapterverbindungen realisiert:

1. **Werte erfassen** – Die beiden Eingabebausteine `InputNumber_I3` und `InputNumber_I4` liefern ihre physikalischen Werte über die Adapterausgänge `rPhys`.
2. **Addition** – Diese Ausgänge werden mit den Adaptereingängen `IN1` bzw. `IN2` des Addierers `AR_ADD_2` verbunden. Der Addierer berechnet die Summe und gibt das Ergebnis an seinem Adapterausgang `OUT` aus.
3. **Wert ausgeben** – Der Ausgang `OUT` von `AR_ADD_2` ist mit dem Adaptereingang `rPhys` des Ausgabebausteins `OutputNumber_N3` verbunden. Dieser stellt das Ergebnis als physikalischen Wert dar.

Die Verbindungen sind:

- `InputNumber_I3.rPhys` ⟶ `AR_ADD_2.IN1`
- `InputNumber_I4.rPhys` ⟶ `AR_ADD_2.IN2`
- `AR_ADD_2.OUT` ⟶ `OutputNumber_N3.rPhys`

Die Übung setzt keine speziellen Vorkenntnisse voraus. Sie kann direkt in der 4diac-IDE durch Erstellen einer neuen Anwendung mit diesem SubApp-Typ und Hinzufügen entsprechender Eingabe-/Ausgabegeräte (z. B. virtuelle Slider oder physische IOs) ausgeführt werden.

Zusammenfassung

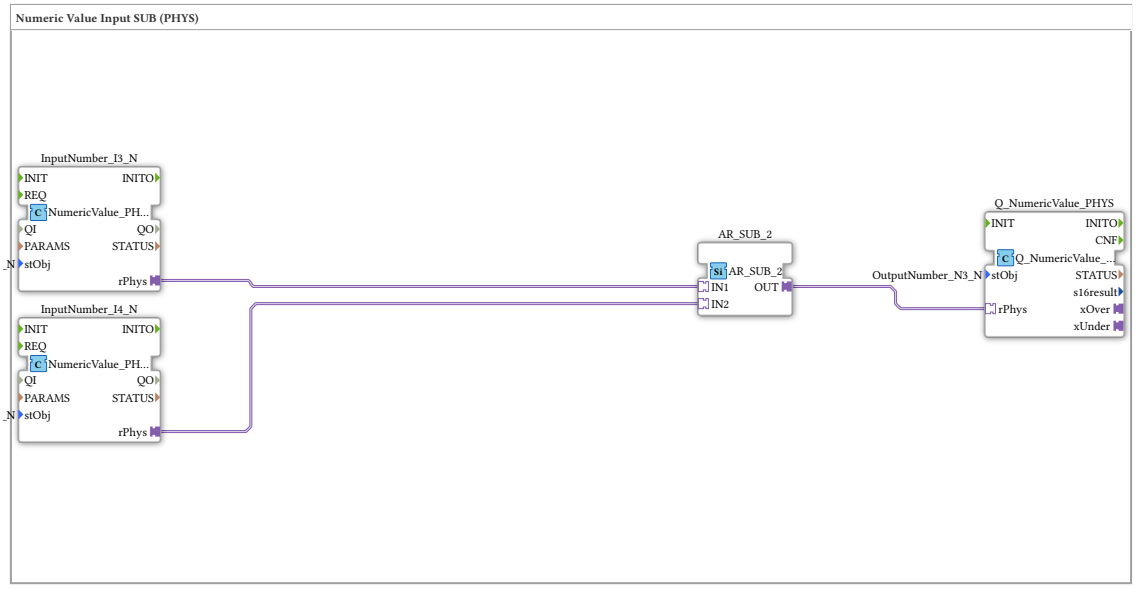
Die SubApp **Uebung_011b3_PHYSA** realisiert eine einfache Addition zweier physikalischer Eingangswerte. Sie zeigt den typischen Aufbau einer Verarbeitungskette mit physikalischen Größen: Eingabe ⟶ Rechenoperation ⟶ Ausgabe. Dabei kommen ausschließlich Adapterverbindungen zum Einsatz, was eine flexible Kopplung der Bausteine ermöglicht. Dies ist die Grundlage für komplexere SPS-Programme mit Einheiten und physikalischen Messwerten in der 4diac-IDE.

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

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

2.1.2.169 Uebung_011b3_PHYS_AX: Numeric Value Input SUB (PHYS)

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_011b3_PHYS_AX (Numeric Value Input SUB (PHYS)).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Numeric Value Input SUB (PHYS)**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_011b3_PHYS_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **InputNumber_I3_N**: Instanz des Typs isobus::UT::io::NumericValue::NumericValue_PHYSA.
  - Parameter stObj = InputNumber_I3_N
- **InputNumber_I4_N**: Instanz des Typs isobus::UT::io::NumericValue::NumericValue_PHYSA.
  - Parameter stObj = InputNumber_I4_N
- **AR_SUB_2**: Instanz des Typs adapter::iec61131::arithmetic::AR_SUB_2.
- **Q_NumericValue_PHYS**: Instanz des Typs isobus::UT::Q::Q_NumericValue_PHYSA.
  - Parameter stObj = OutputNumber_N3_N

Verbindungen und Schnittstellen

Adapterverbindungen:

- InputNumber_I3_N.rPhys -> AR_SUB_2.IN1
- InputNumber_I4_N.rPhys -> AR_SUB_2.IN2
- AR_SUB_2.OUT -> Q_NumericValue_PHYS.rPhys

Programmablauf und Funktionsweise

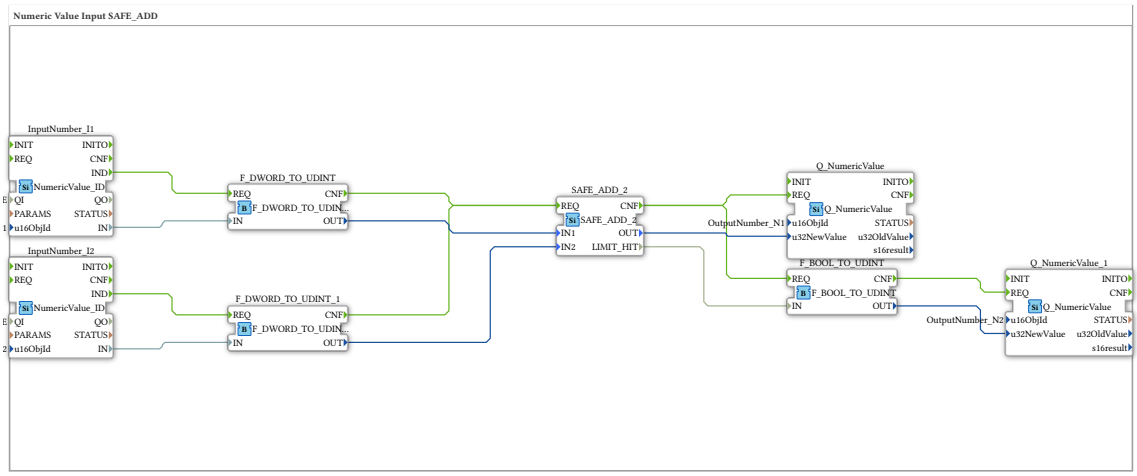
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_011b3_PHYS_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.170 Uebung_011b4_AX: Numeric Value Input SAFE_ADD

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_011b4_AX (Numeric Value Input SAFE_ADD).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Numeric Value Input SAFE_ADD**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_011b4_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **InputNumber_I1**: Instanz des Typs isobus::UT::io::NumericValue::NumericValue_ID.
  - Parameter QI = TRUE
  - Parameter u16ObjId = InputNumber_I1
- **InputNumber_I2**: Instanz des Typs isobus::UT::io::NumericValue::NumericValue_ID.
  - Parameter QI = TRUE
  - Parameter u16ObjId = InputNumber_I2
- **F_DWORD_TO_UDINT**: Instanz des Typs iec61131::conversion::F_DWORD_TO_UDINT.
- **F_DWORD_TO_UDINT_1**: Instanz des Typs iec61131::conversion::F_DWORD_TO_UDINT.
- **SAFE_ADD_2**: Instanz des Typs SafeArithmetic::arithmetic::SAFE_ADD_2.
- **Q_NumericValue**: Instanz des Typs isobus::UT::Q::Q_NumericValue.
  - Parameter u16ObjId = OutputNumber_N1
- **F_BOOL_TO_UDINT**: Instanz des Typs iec61131::conversion::F_BOOL_TO_UDINT.
- **Q_NumericValue_1**: Instanz des Typs isobus::UT::Q::Q_NumericValue.
  - Parameter u16ObjId = OutputNumber_N2

Verbindungen und Schnittstellen

Ereignisverbindungen:

- InputNumber_I1.IND -> F_DWORD_TO_UDINT.REQ
- InputNumber_I2.IND -> F_DWORD_TO_UDINT_1.REQ
- F_DWORD_TO_UDINT.CNF -> SAFE_ADD_2.REQ
- F_DWORD_TO_UDINT_1.CNF -> SAFE_ADD_2.REQ
- SAFE_ADD_2.CNF -> Q_NumericValue.REQ
- SAFE_ADD_2.CNF -> F_BOOL_TO_UDINT.REQ
- F_BOOL_TO_UDINT.CNF -> Q_NumericValue_1.REQ

Datenverbindungen:

- InputNumber_I1.IN -> F_DWORD_TO_UDINT.IN
- InputNumber_I2.IN -> F_DWORD_TO_UDINT_1.IN
- F_DWORD_TO_UDINT.OUT -> SAFE_ADD_2.IN1
- F_DWORD_TO_UDINT_1.OUT -> SAFE_ADD_2.IN2

- SAFE_ADD_2.OUT -> Q_NumericValue.u32NewValue
- SAFE_ADD_2.LIMIT_HIT -> F_BOOL_TO_UDINT.IN
- F_BOOL_TO_UDINT.OUT -> Q_NumericValue_1.u32NewValue

Programmablauf und Funktionsweise

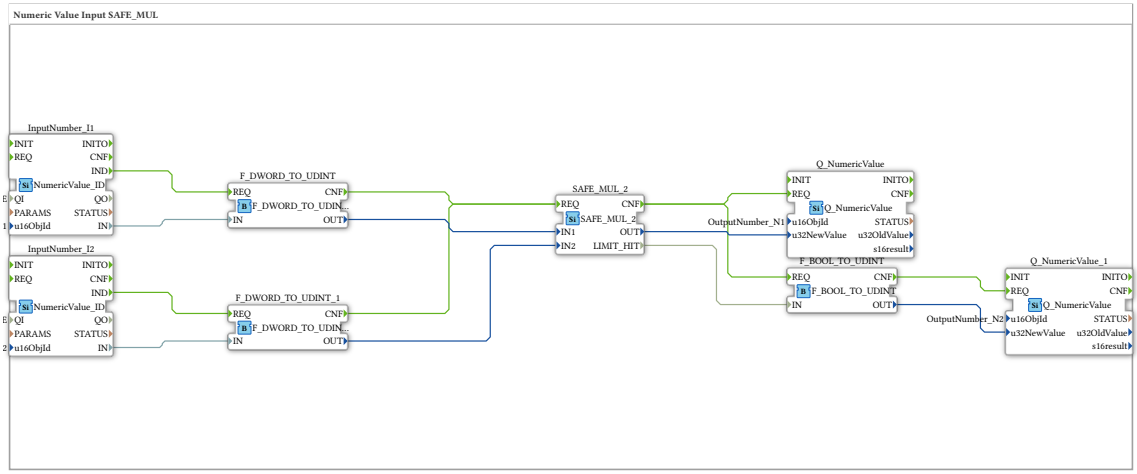
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_011b4_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.171 Uebung_011b5_AX: Numeric Value Input SAFE_MUL

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_011b5_AX (Numeric Value Input SAFE_MUL).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Numeric Value Input SAFE_MUL**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_011b5_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **InputNumber_I1**: Instanz des Typs isobus::UT::io::NumericValue::NumericValue_ID.
  - Parameter QI = TRUE
  - Parameter u16ObjId = InputNumber_I1
- **InputNumber_I2**: Instanz des Typs isobus::UT::io::NumericValue::NumericValue_ID.
  - Parameter QI = TRUE
  - Parameter u16ObjId = InputNumber_I2
- **F_DWORD_TO_UDINT**: Instanz des Typs iec61131::conversion::F_DWORD_TO_UDINT.
- **F_DWORD_TO_UDINT_1**: Instanz des Typs iec61131::conversion::F_DWORD_TO_UDINT.
- **SAFE_MUL_2**: Instanz des Typs SafeArithmetic::arithmetic::SAFE_MUL_2.
- **Q_NumericValue**: Instanz des Typs isobus::UT::Q::Q_NumericValue.
  - Parameter u16ObjId = OutputNumber_N1
- **F_BOOL_TO_UDINT**: Instanz des Typs iec61131::conversion::F_BOOL_TO_UDINT.
- **Q_NumericValue_1**: Instanz des Typs isobus::UT::Q::Q_NumericValue.

– Parameter u16ObjId = OutputNumber_N2

Verbindungen und Schnittstellen

Ereignisverbindungen:

- InputNumber_I1.IND -> F_DWORD_TO_UDINT.REQ
- InputNumber_I2.IND -> F_DWORD_TO_UDINT_1.REQ
- F_DWORD_TO_UDINT.CNF -> SAFE_MUL_2.REQ
- F_DWORD_TO_UDINT_1.CNF -> SAFE_MUL_2.REQ
- SAFE_MUL_2.CNF -> Q_NumericValue.REQ
- SAFE_MUL_2.CNF -> F_BOOL_TO_UDINT.REQ
- F_BOOL_TO_UDINT.CNF -> Q_NumericValue_1.REQ

Datenverbindungen:

- InputNumber_I1.IN -> F_DWORD_TO_UDINT.IN
- InputNumber_I2.IN -> F_DWORD_TO_UDINT_1.IN
- F_DWORD_TO_UDINT.OUT -> SAFE_MUL_2.IN1
- F_DWORD_TO_UDINT_1.OUT -> SAFE_MUL_2.IN2
- SAFE_MUL_2.OUT -> Q_NumericValue.u32NewValue
- SAFE_MUL_2.LIMIT_HIT -> F_BOOL_TO_UDINT.IN
- F_BOOL_TO_UDINT.OUT -> Q_NumericValue_1.u32NewValue

Programmablauf und Funktionsweise

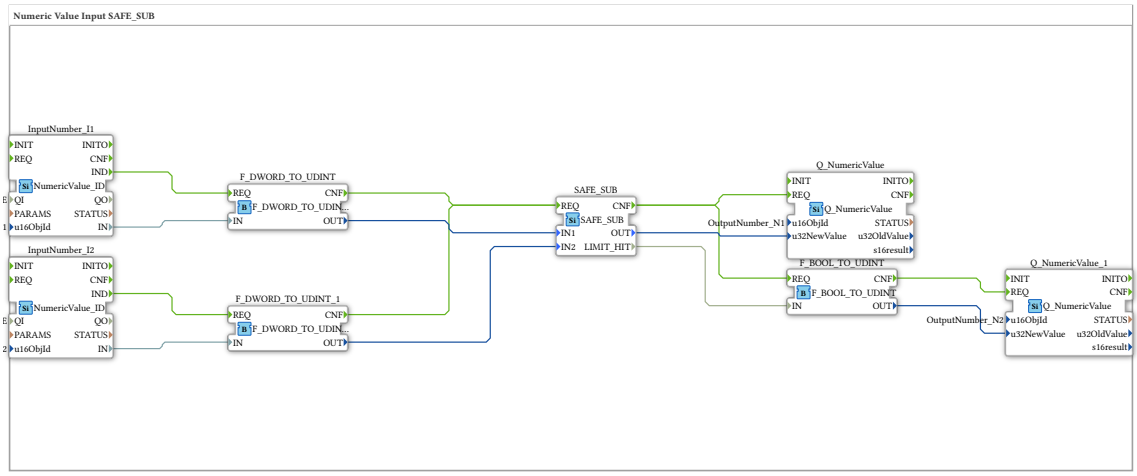
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_011b5_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.172 Uebung_011b6_AX: Numeric Value Input SAFE_SUB

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_011b6_AX (Numeric Value Input SAFE_SUB).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Numeric Value Input SAFE_SUB**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_011b6_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **InputNumber_I1**: Instanz des Typs isobus::UT::io::NumericValue::NumericValue_ID.
  - Parameter QI = TRUE
  - Parameter u16ObjId = InputNumber_I1
- **InputNumber_I2**: Instanz des Typs isobus::UT::io::NumericValue::NumericValue_ID.
  - Parameter QI = TRUE
  - Parameter u16ObjId = InputNumber_I2
- **F_DWORD_TO_UDINT**: Instanz des Typs iec61131::conversion::F_DWORD_TO_UDINT.
- **F_DWORD_TO_UDINT_1**: Instanz des Typs iec61131::conversion::F_DWORD_TO_UDINT.
- **SAFE_SUB**: Instanz des Typs SafeArithmetic::arithmetic::SAFE_SUB.
- **Q_NumericValue**: Instanz des Typs isobus::UT::Q::Q_NumericValue.
  - Parameter u16ObjId = OutputNumber_N1
- **F_BOOL_TO_UDINT**: Instanz des Typs iec61131::conversion::F_BOOL_TO_UDINT.
- **Q_NumericValue_1**: Instanz des Typs isobus::UT::Q::Q_NumericValue.
  - Parameter u16ObjId = OutputNumber_N2

Verbindungen und Schnittstellen

Ereignisverbindungen:

- InputNumber_I1.IND -> F_DWORD_TO_UDINT.REQ
- InputNumber_I2.IND -> F_DWORD_TO_UDINT_1.REQ
- F_DWORD_TO_UDINT.CNF -> SAFE_SUB.REQ
- F_DWORD_TO_UDINT_1.CNF -> SAFE_SUB.REQ
- SAFE_SUB.CNF -> Q_NumericValue.REQ
- SAFE_SUB.CNF -> F_BOOL_TO_UDINT.REQ
- F_BOOL_TO_UDINT.CNF -> Q_NumericValue_1.REQ

Datenverbindungen:

- InputNumber_I1.IN -> F_DWORD_TO_UDINT.IN
- InputNumber_I2.IN -> F_DWORD_TO_UDINT_1.IN
- F_DWORD_TO_UDINT.OUT -> SAFE_SUB.IN1
- F_DWORD_TO_UDINT_1.OUT -> SAFE_SUB.IN2
- SAFE_SUB.OUT -> Q_NumericValue.u32NewValue
- SAFE_SUB.LIMIT_HIT -> F_BOOL_TO_UDINT.IN
- F_BOOL_TO_UDINT.OUT -> Q_NumericValue_1.u32NewValue

Programmablauf und Funktionsweise

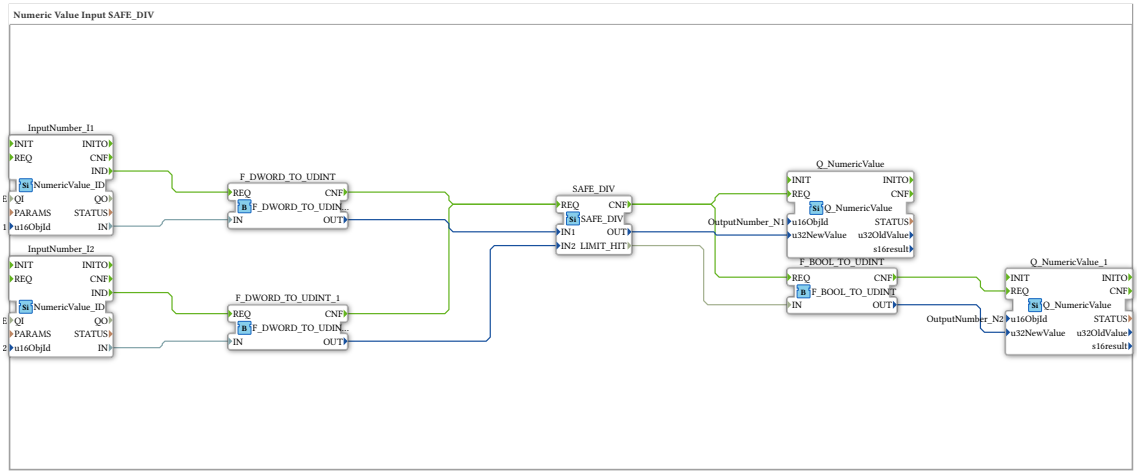
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_011b6_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.173 Uebung_011b7_AX: Numeric Value Input SAFE_DIV

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_011b7_AX (Numeric Value Input SAFE_DIV).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Numeric Value Input SAFE_DIV**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_011b7_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **InputNumber_I1**: Instanz des Typs isobus::UT::io::NumericValue::NumericValue_ID.
  - Parameter QI = TRUE
  - Parameter u16ObjId = InputNumber_I1
- **InputNumber_I2**: Instanz des Typs isobus::UT::io::NumericValue::NumericValue_ID.
  - Parameter QI = TRUE
  - Parameter u16ObjId = InputNumber_I2
- **F_DWORD_TO_UDINT**: Instanz des Typs iec61131::conversion::F_DWORD_TO_UDINT.
- **F_DWORD_TO_UDINT_1**: Instanz des Typs iec61131::conversion::F_DWORD_TO_UDINT.
- **SAFE_DIV**: Instanz des Typs SafeArithmetic::arithmetic::SAFE_DIV.
- **Q_NumericValue**: Instanz des Typs isobus::UT::Q::Q_NumericValue.
  - Parameter u16ObjId = OutputNumber_N1
- **F_BOOL_TO_UDINT**: Instanz des Typs iec61131::conversion::F_BOOL_TO_UDINT.
- **Q_NumericValue_1**: Instanz des Typs isobus::UT::Q::Q_NumericValue.
  - Parameter u16ObjId = OutputNumber_N2

Verbindungen und Schnittstellen

Ereignisverbindungen:

- InputNumber_I1.IND -> F_DWORD_TO_UDINT.REQ
- InputNumber_I2.IND -> F_DWORD_TO_UDINT_1.REQ
- F_DWORD_TO_UDINT.CNF -> SAFE_DIV.REQ
- F_DWORD_TO_UDINT_1.CNF -> SAFE_DIV.REQ
- SAFE_DIV.CNF -> Q_NumericValue.REQ
- SAFE_DIV.CNF -> F_BOOL_TO_UDINT.REQ
- F_BOOL_TO_UDINT.CNF -> Q_NumericValue_1.REQ

Datenverbindungen:

- InputNumber_I1.IN -> F_DWORD_TO_UDINT.IN
- InputNumber_I2.IN -> F_DWORD_TO_UDINT_1.IN
- F_DWORD_TO_UDINT.OUT -> SAFE_DIV.IN1
- F_DWORD_TO_UDINT_1.OUT -> SAFE_DIV.IN2
- SAFE_DIV.OUT -> Q_NumericValue.u32NewValue
- SAFE_DIV.LIMIT_HIT -> F_BOOL_TO_UDINT.IN
- F_BOOL_TO_UDINT.OUT -> Q_NumericValue_1.u32NewValue

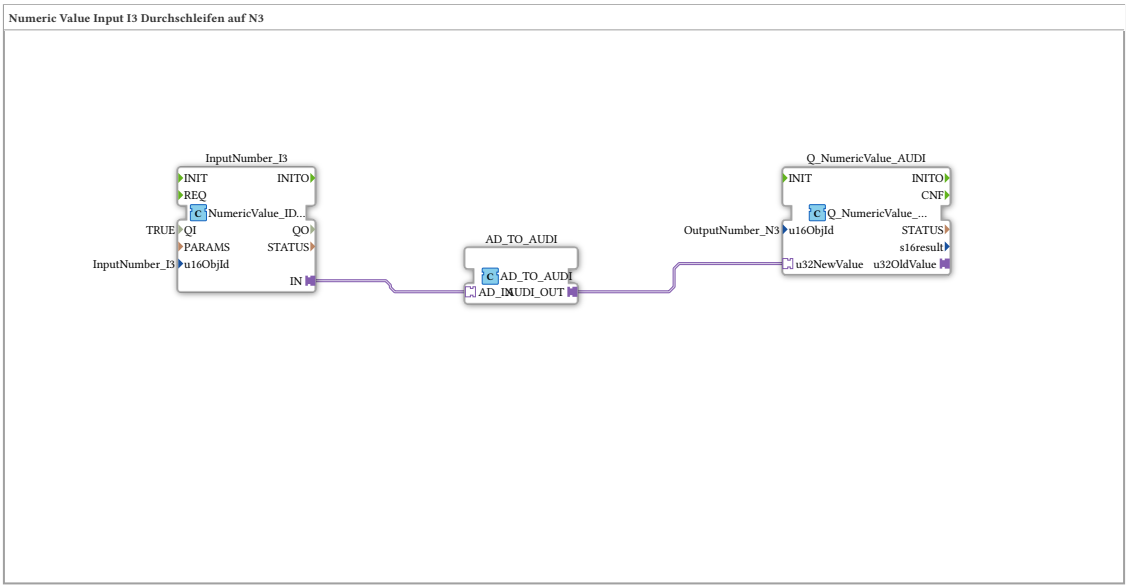
Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_011b7_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.174 Uebung_011c_AUDI: Numeric Value Input I3 Durchschleifen auf N3



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung demonstriert das Durchschleifen eines numerischen Werts von einem Eingangsbaustein (`InputNumber_I3`) auf einen Ausgangsbaustein (`Q_NumericValue_AUDI`) unter Verwendung eines Adapterbausteins (`AD_TO_AUDI`). Der Wert wird dabei unverändert übertragen („Durchschleifen“). Die SubApplikation ist als wiederverwendbare Komponente für ISOBUS-Anwendungen konzipiert.

Verwendete Funktionsbausteine (FBs)

- **InputNumber_I3**
  - **Typ**: `isobus::UT::io::NumericValue::NumericValue_IDA`
  - **Parameter**:
    - `QI` = `TRUE`
    - `u16ObjId` = `InputNumber_I3`
  - **Funktion**: Stellt einen numerischen Eingangswert (z.B. von einem Bedienelement) über einen Adapterausgang (`IN`) bereit.
- **AD_TO_AUDI**
  - **Typ**: `adapter::conversion::unidirectional::AD_TO_AUDI`
  - **Parameter**: Keine
  - **Funktion**: Wandelt das Adaptersignal des Eingangs (`AD_IN`) in ein für den Ausgangsbaustein passendes Signal (`AUDI_OUT`). In dieser Übung wird der Wert unverändert durchgereicht.
- **Q_NumericValue_AUDI**
  - **Typ**: `isobus::UT::Q::Q_NumericValue_AUDI`

- **Parameter**:
  - `u16ObjId` = `OutputNumber_N3`
- **Funktion**: Nimmt den numerischen Wert über den Dateneingang `u32NewValue` entgegen und stellt ihn als ISOBUS-Ausgangsobjekt (z. B. zur Anzeige) zur Verfügung.

Programmablauf und Verbindungen

1. Der Eingangsbaustein `InputNumber_I3` liefert den aktuellen numerischen Wert an seinem Adapterausgang `IN`.
2. Dieser Wert wird über eine Adapterverbindung an den Eingang `AD_IN` des Bausteins `AD_TO_AUDI` weitergeleitet.
3. Der Adapter `AD_TO_AUDI` reicht den empfangenen Wert unverändert an seinen Ausgang `AUDI_OUT` weiter.
4. Der Wert gelangt über die zweite Adapterverbindung an den Dateneingang `u32NewValue` des Ausgangsbausteins `Q_NumericValue_AUDI`.
5. Der Ausgangsbaustein stellt den Wert anschließend als ISOBUS-Ausgangsobjekt `OutputNumber_N3` bereit.

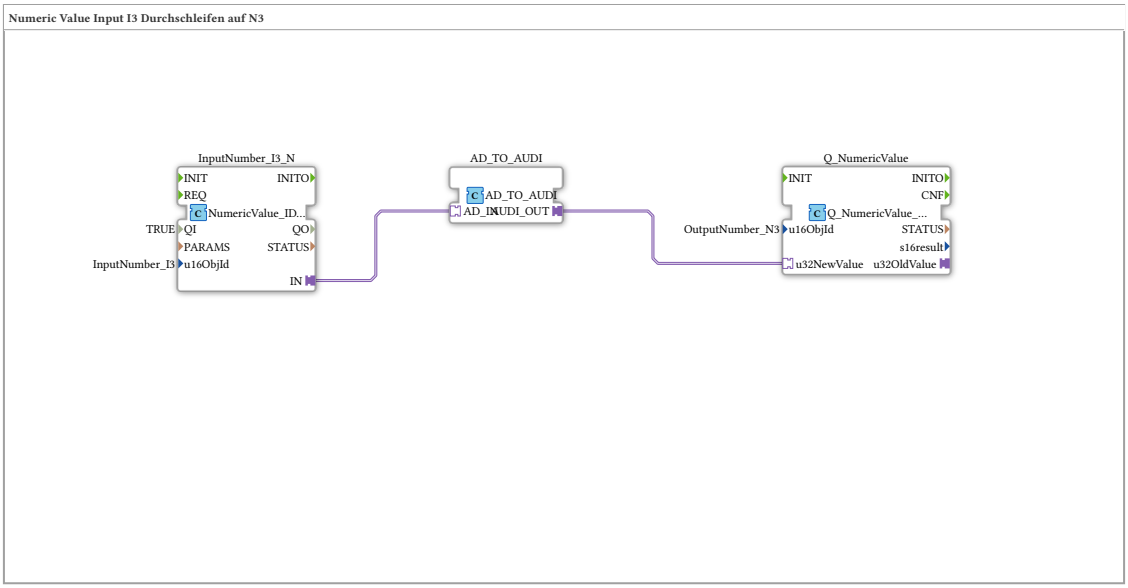
Die gesamte Datenübertragung erfolgt ereignisgesteuert – sobald sich der Eingangswert ändert, wird die Kette automatisch durchlaufen.

Zusammenfassung

Die Übung zeigt das grundlegende Prinzip des Datenaustauschs zwischen ISOBUS-Ein- und Ausgangsbausteinen mithilfe eines Adapters. Sie vermittelt den Aufbau einer einfachen Durchschleif-Logik und bereitet auf komplexere Verarbeitungsschritte in späteren Übungen vor. Der Schwerpunkt liegt auf dem Verständnis von Adapterverbindungen und der Parametrierung der NumericValue-Bausteine.

2.1.2.175 Uebung_011c_AX: Numeric Value Input I3 Durchschleifen auf N3

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_011c_AX (Numeric Value Input I3 Durchschleifen auf N3).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Numeric Value Input I3 Durchschleifen auf N3**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_011c_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **InputNumber_I3_N**: Instanz des Typs isobus::UT::io::NumericValue::NumericValue_IDA.
  - Parameter QI = TRUE
  - Parameter u16ObjId = InputNumber_I3
- **AD_TO_AUDI**: Instanz des Typs adapter::conversion::unidirectional::AD_TO_AUDI.
- **Q_NumericValue**: Instanz des Typs isobus::UT::Q::Q_NumericValue_AUDI.
  - Parameter u16ObjId = OutputNumber_N3

Verbindungen und Schnittstellen

Adapterverbindungen:

- InputNumber_I3_N.IN -> AD_TO_AUDI.AD_IN
- AD_TO_AUDI.AUDI_OUT -> Q_NumericValue.u32NewValue

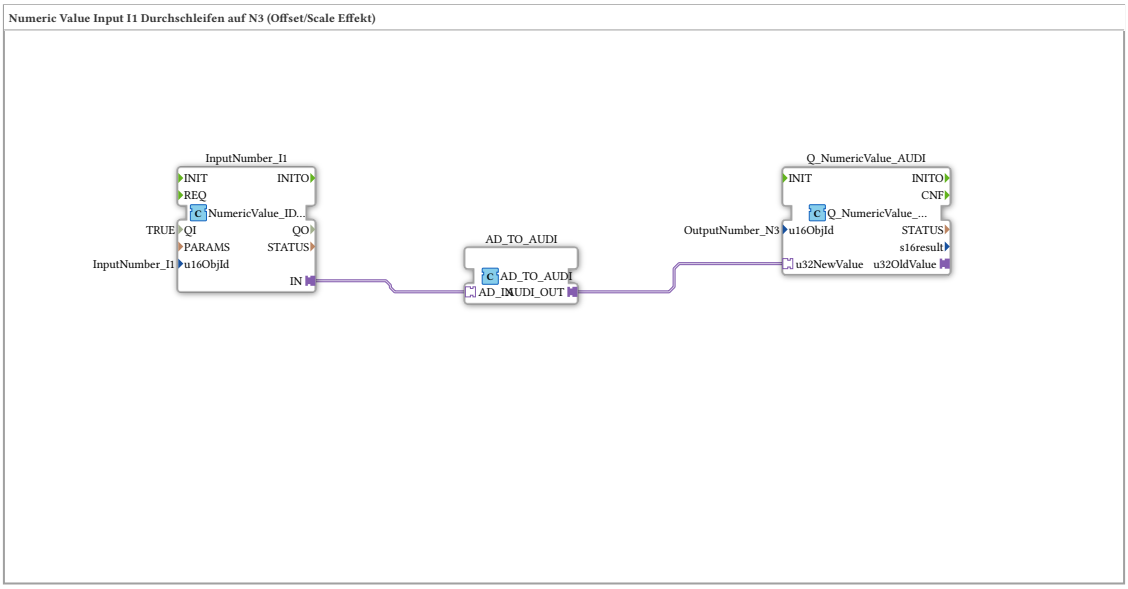
Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_011c_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.176 Uebung_011d_AUDI: Numeric Value Input I1 Durchschleifen auf N3 (Offset/ Scale Effekt)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert das Durchschleifen eines numerischen Eingangswerts (I1) auf einen Ausgang (N3) unter Verwendung eines Offset-/Scale-Effekts. Der eingehende Wert wird über einen Adapter gewandelt und an einen Ausgangs-Funktionsbaustein weitergegeben. Beispiel:

- Eingabe von 100000 auf I1 ⟶ N3 zeigt 0,00
- Eingabe von 50000 auf I1 ⟶ N3 zeigt −500,00

Verwendete Funktionsbausteine (FBs)

- **InputNumber_I1**

  - **Typ**: `isobus::UT::io::NumericValue::NumericValue_IDA`
  - **Parameter**:
    - `QI` = TRUE
    - `u16ObjId` = InputNumber_I1
  - **Funktion**: Liest den numerischen Wert des ISOBUS-Objekts „InputNumber_I1“ und stellt ihn über den Adapterausgang `IN` bereit.

- **AD_TO_AUDI**
  - **Typ**: `adapter::conversion::unidirectional::AD_TO_AUDI`
  - **Parameter**: keine expliziten Parameter
  - **Funktion**: Wandelt den ankommenden Adapterwert (AD) in einen AUDI-kompatiblen Wert um. Dabei wird der Eingangswert mit einem internen Offset und Skalierungsfaktor verarbeitet, um den gewünschten Effekt zu erzielen.

- **Q_NumericValue_AUDI**
  - **Typ**: `isobus::UT::Q::Q_NumericValue_AUDI`
  - **Parameter**:
    - `u16ObjId` = OutputNumber_N3
  - **Funktion**: Empfängt den konvertierten Wert über den Datenport `u32NewValue` und schreibt ihn auf das ISOBUS-Ausgangsobjekt „OutputNumber_N3“.

Programmablauf und Verbindungen

1. Der Funktionsbaustein `InputNumber_I1` erfasst den aktuellen Wert des ISOBUS-Eingangsobjekts.
2. Über die **Adapterverbindung** wird der Wert von `InputNumber_I1.IN` an `AD_TO_AUDI.AD_IN` weitergeleitet.
3. Im Baustein `AD_TO_AUDI` erfolgt die Umrechnung (Offset/Skalierung) des Wertes.
4. Der berechnete AUDI-Wert verlässt den Baustein über den Ausgang `AUDI_OUT` und wird an den Dateneingang `u32NewValue` von `Q_NumericValue_AUDI` geführt.
5. `Q_NumericValue_AUDI` schreibt den finalen Wert in das Ausgangsobjekt „OutputNumber_N3“.

Die gesamte Logik ist als Subapplikation realisiert und verwendet keine weiteren Unterbausteine. Der Ablauf ist rein datengetrieben – sobald sich der Eingangswert ändert, wird die gesamte Kette durchlaufen.

Zusammenfassung

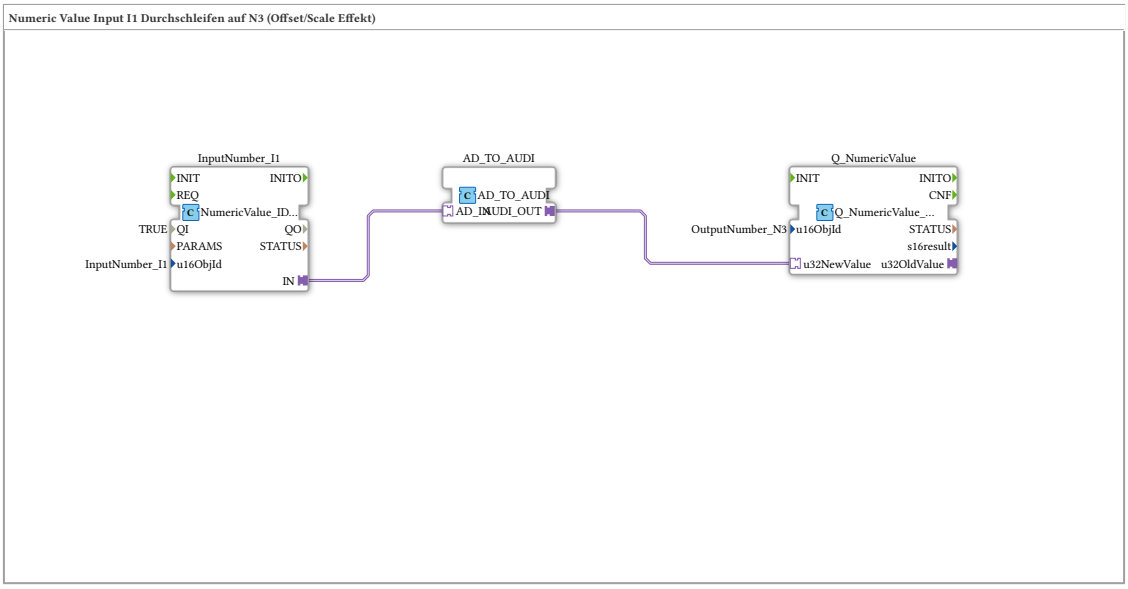
Die Übung „Uebung_011d_AUDI“ veranschaulicht, wie ein numerischer ISOBUS-Eingangswert über einen Adapterbaustein mit Offset/Skalierung in einen Ausgangswert umgesetzt wird. Sie schult den Umgang mit Adapterverbindungen, der Parametrierung von ISOBUS-Objekten und dem Verständnis von Skalierungseffekten in der 4diac-IDE.

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

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

2.1.2.177 Uebung_011d_AX: Numeric Value Input I1 Durchschleifen auf N3 (Offset/Scale Effekt)

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_011d_AX (Numeric Value Input I1 Durchschleifen auf N3 (Offset/Scale Effekt)).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Numeric Value Input I1 Durchschleifen auf N3 (Offset/Scale Effekt)**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_011d_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **InputNumber_I1**: Instanz des Typs isobus::UT::io::NumericValue::NumericValue_IDA.
  - Parameter QI = TRUE
  - Parameter u16ObjId = InputNumber_I1
- **AD_TO_AUDI**: Instanz des Typs adapter::conversion::unidirectional::AD_TO_AUDI.
- **Q_NumericValue**: Instanz des Typs isobus::UT::Q::Q_NumericValue_AUDI.
  - Parameter u16ObjId = OutputNumber_N3

Verbindungen und Schnittstellen

Adapterverbindungen:

- InputNumber_I1.IN -> AD_TO_AUDI.AD_IN
- AD_TO_AUDI.AUDI_OUT -> Q_NumericValue.u32NewValue

Hinweise aus dem Modell

Beispiel: I1-Eingabe 100000 ⟶ N3 zeigt 0.00, Eingabe 50000 ⟶ N3 zeigt −500.00.

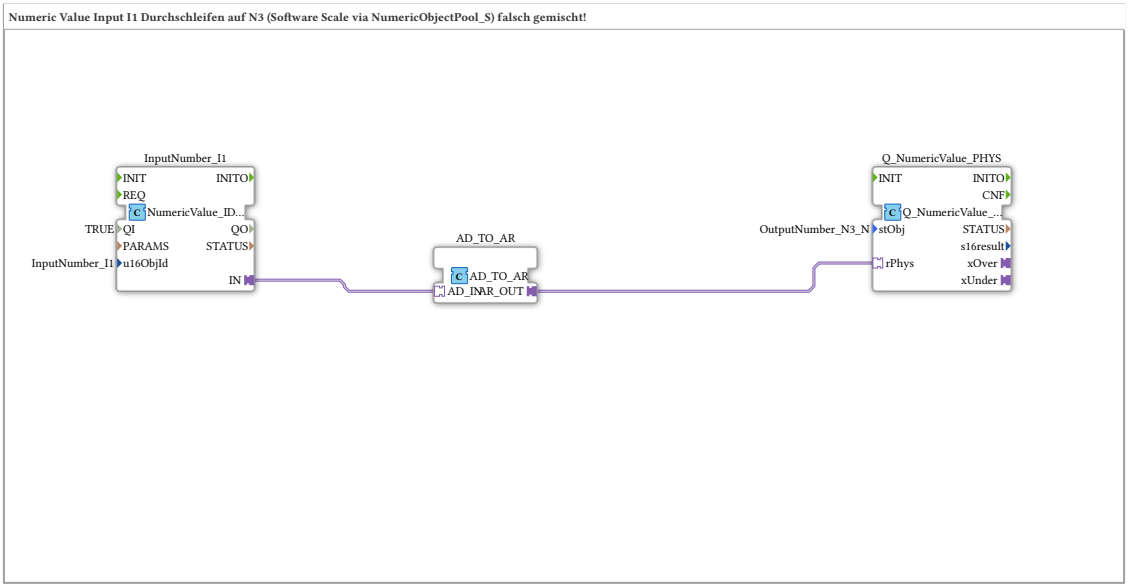
Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_011d_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.178 Uebung_011e_MIXA: Numeric Value Input I1 Durchschleifen auf N3 (Software Scale via NumericObjectPool_S) falsch gemischt



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung demonstriert das fehlerhafte Durchschleifen eines numerischen Eingabewerts von **InputNumber_I1** auf **OutputNumber_N3**. Der Wert wird ohne korrekte Skalierung (Software Scale) weitergegeben, da die verwendeten Namespaces `DefaultPool` und `DefaultPool_Numeric` inkompatibel sind. Ziel ist es, das Problembewusstsein für die korrekte Zuordnung von Quell- und Zielobjekten zu schärfen.

Verwendete Funktionsbausteine (FBs)

InputNumber_I1

- **Typ**: `isobus::UT::io::NumericValue::NumericValue_IDA`
- **Parameter**:
  - `QI` = `TRUE`
  - `u16ObjId` = `InputNumber_I1`
- **Funktionsweise**:
  Liest den aktuellen Wert des numerischen Eingangs **I1** aus dem Pool `DefaultPool` und stellt ihn am Adapterausgang `IN` bereit.

F_DWORD_TO_UDINT

- **Typ**: `adapter::conversion::unidirectional::AD_TO_AR`
- **Parameter**: keine
- **Funktionsweise**:
  Konvertiert den am Adaptereingang `AD_IN` anliegenden DWORD-Wert in einen UDINT-Wert und gibt diesen am Adapterausgang `AR_OUT` aus.

Q_NumericValue_PHYS

- **Typ**: `isobus::UT::Q::Q_NumericValue_PHYSA`
- **Parameter**:
  - `stObj` = `OutputNumber_N3`
- **Funktionsweise**:
  Nimmt den konvertierten Wert über den Adaptereingang `rPhys` entgegen und schreibt ihn auf den numerischen Ausgang **N3** des Pools `DefaultPool_Numeric`.

Programmablauf und Verbindungen

Das SubApp-Netzwerk verbindet die drei FB in einer Kette:

1. **InputNumber_I1** → liefert den aktuellen Wert von I1 als DWORD an seinem Adapterausgang `IN`.
2. **F_DWORD_TO_UDINT** → empfängt den DWORD-Wert am `AD_IN`, konvertiert ihn in einen UDINT und gibt diesen an `AR_OUT`.
3. **Q_NumericValue_PHYS** → erhält den UDINT-Wert an `rPhys` und schreibt ihn auf das Ausgangsobjekt `OutputNumber_N3`.

Hinweis: Die Namespaces der beiden Objekte sind inkompatibel:

- `InputNumber_I1` stammt aus `Uebungen::const::UT::DefaultPool`.
- `OutputNumber_N3` stammt aus `Uebungen::const::UT::DefaultPool_Numeric`.

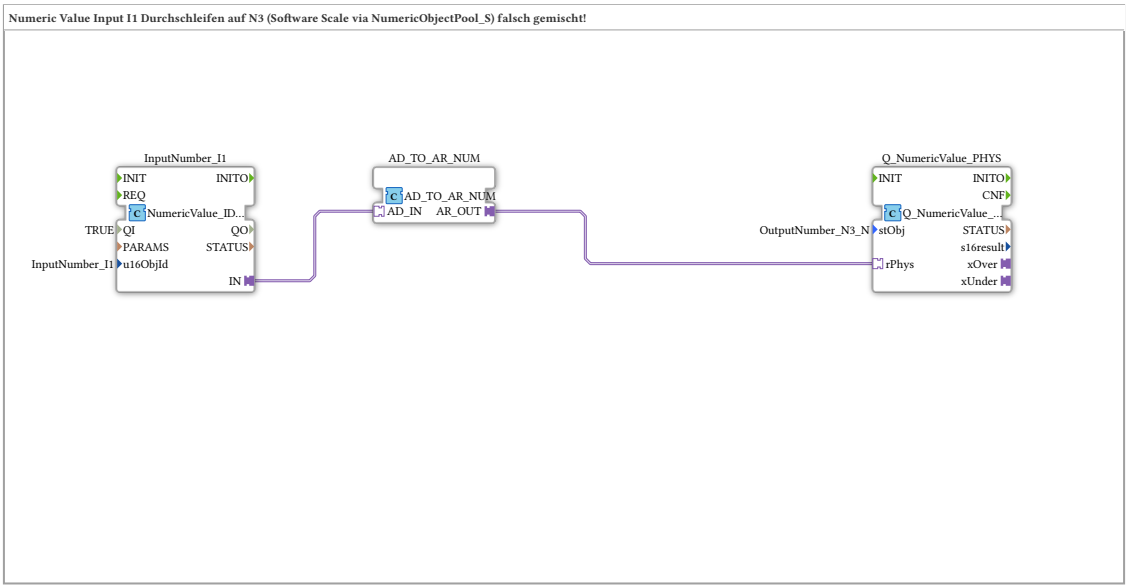
Daher wird der Wert zwar technisch weitergeleitet, aber die Skalierung bzw. Objektverknüpfung (Software Scale) ist nicht korrekt eingerichtet – die Übung dient als Negativbeispiel.

Zusammenfassung

Die Übung zeigt eine bewusst falsch gemischte Konfiguration, bei der ein Eingangswert ohne Anpassung der Skalierung auf einen Ausgang eines anderen Namespace durchgeschleift wird. Der erwartete Effekt: Der Wert wird angezeigt (z. B. 10.00), aber die zugrunde liegende Objektzuordnung ist inkonsistent. Dies verdeutlicht die Notwendigkeit, Quell- und Zielobjekte aus demselben Pool zu verwenden oder eine explizite Skalierung vorzunehmen.

2.1.2.179 Uebung_011e_MIX_AX: Numeric Value Input I1 Durchschleifen auf N3 (Software Scale via NumericObjectPool_S) falsch gemischt

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_011e_MIX_AX (Numeric Value Input I1 Durchschleifen auf N3 (Software Scale via NumericObjectPool_S) falsch gemischt!).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Numeric Value Input I1 Durchschleifen auf N3 (Software Scale via NumericObjectPool_S) falsch gemischt!**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_011e_MIX_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **InputNumber_I1**: Instanz des Typs isobus::UT::io::NumericValue::NumericValue_IDA.
  - Parameter QI = TRUE
  - Parameter u16ObjId = InputNumber_I1
- **AD_TO_AR_NUM**: Instanz des Typs adapter::conversion::unidirectional::AD_TO_AR_NUM.
- **Q_NumericValue_PHYS**: Instanz des Typs isobus::UT::Q::Q_NumericValue_PHYSA.

– Parameter stObj = OutputNumber_N3_N

Verbindungen und Schnittstellen

Adapterverbindungen:

- InputNumber_I1.IN -> AD_TO_AR_NUM.AD_IN
- AD_TO_AR_NUM.AR_OUT -> Q_NumericValue_PHYS.rPhys

Hinweise aus dem Modell

Beispiel: I1-Eingabe 10 ⟶ F_RAW_TO_PHYS(I1) ⟶ 10.0 ⟶ Q_NumericValue_PHYS(N3) ⟶ N3 zeigt 10.00.

die beiden Namespaces sind INKOMPATIBEL !!!

Uebungen::const::UT::DefaultPool::InputNumber_I1
Uebungen::const::UT::DefaultPool_Numeric::OutputNumber_N3_N

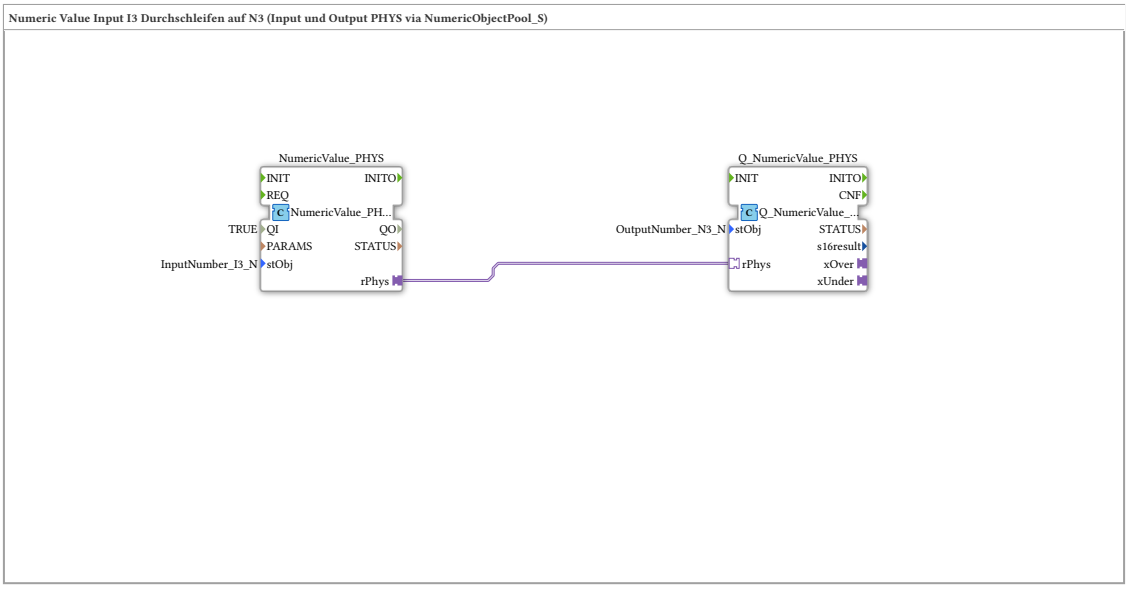
Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_011e_MIX_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.180 Uebung_011f_AX: Numeric Value Input I3 durchschleifen auf N3 (Input und Output PHYS via NumericObjectPool_S)



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung schleift einen numerischen Eingabewert des VT (Eingabefeld `I3`) unverändert auf ein numerisches Ausgabefeld (`N3`) durch. Eingang und Ausgang sind dabei jeweils als physikalischer `REAL`-Wert (`PHYS`) ausgeführt: Die Umrechnung zwischen Rohwert und physikalischem Wert (Skalierung, Offset, Dezimalstellen) steckt in den Parametern vom Typ `NumericObjectPool_S` (`stObj`), die aus der Pool-Konstantendatei importiert werden. Die SubApp hat keine eigene Schnittstelle.

Verwendete Funktionsbausteine (FBs)

- **NumericValue_PHYS**: Numerischer VT-Eingabewert als physikalischer Wert
  - **Typ**: `isobus::UT::io::NumericValue::NumericValue_PHYSA`
  - **Parameter**: `QI = TRUE`, `stObj = InputNumber_I3_N`
  - **Erklärung**: Liefert den im VT-Eingabefeld `I3` eingegebenen Wert als physikalischen `REAL`-Wert über den Adapter-Ausgang `rPhys` (AR-Adapter).
- **Q_NumericValue_PHYS**: Numerischer VT-Ausgabewert aus einem physikalischen Wert
  - **Typ**: `isobus::UT::Q::Q_NumericValue_PHYSA`
  - **Parameter**: `stObj = OutputNumber_N3_N`
  - **Erklärung**: Empfängt einen physikalischen `REAL`-Wert über den Adapter `rPhys` und schreibt ihn in das numerische Ausgabeobjekt `N3` des VT.

Die Konstanten `InputNumber_I3_N` und `OutputNumber_N3_N` stammen aus `Uebungen::const::UT::DefaultPool_Numeric`.

Programmablauf und Verbindungen

1. `NumericValue_PHYS.rPhys` → `Q_NumericValue_PHYS.rPhys` (AdapterConnection): der physikalische Wert der Eingabe `I3` wird direkt an die Ausgabe `N3` weitergegeben.

Beispiel (Kommentar im Modell): Eine Eingabe von −500.00 in `I3` ergibt `rPhys = -500.0`; `Q_NumericValue_PHYS` (`N3`) zeigt anschließend −500.00 an.

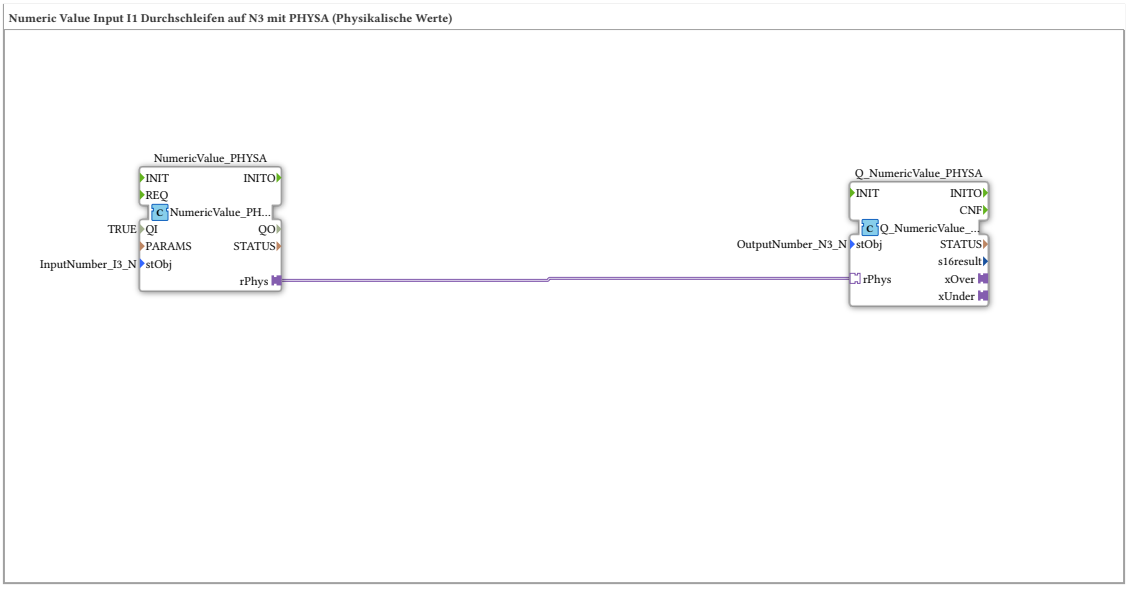
Zusammenfassung

Die Übung `Uebung_011f_AX` zeigt die einfachste Verwendung der PHYS-Varianten für numerische VT-Objekte: Ein Eingabe-FB und ein Ausgabe-FB werden über eine einzige AR-Adapterverbindung gekoppelt, die Skalierung und der Offset stecken vollständig in den `NumericObjectPool_S`-Parametern.

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

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

2.1.2.181 Uebung_011f_PHYSA: Numeric Value Input I1 Durchschleifen auf N3 mit PHYSA (Physikalische Werte)



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung demonstriert das direkte Durchschleifen eines physikalischen Werts (PHYSA) ohne Konvertierung. Ein eingehender numerischer Wert von der Schnittstelle `InputNumber_I3` wird unverändert an die Ausgangsschnittstelle `OutputNumber_N3` weitergegeben. Die Übung veranschaulicht den Umgang mit physikalischen Werten im ISO-BUS-Kontext und die einfache Verbindung von Ein- und Ausgabebausteinen.

Verwendete Funktionsbausteine (FBs)

In der SubApp sind zwei vordefinierte Funktionsbausteine eingesetzt:

NumericValue_PHYSA

- **Typ**: `isobus::UT::io::NumericValue::NumericValue_PHYSA`
- **Parameter**:
  - `QI` = `TRUE` (Freigabe des Bausteins)
  - `stObj` = `InputNumber_I3` (Referenz auf die Eingabestelle)
- **Adapterausgang**: `rPhys` (physikalischer Wert)

Dieser Baustein liest den physikalischen Wert von der Eingabeadresse `InputNumber_I3`. Er ist so konfiguriert, dass der Wert direkt (ohne Umrechnung) am Adapterausgang `rPhys` bereitgestellt wird.

Q_NumericValue_PHYSA

- **Typ**: `isobus::UT::Q::Q_NumericValue_PHYSA`
- **Parameter**:
  - `stObj` = `OutputNumber_N3` (Referenz auf die Ausgabestelle)
- **Adaptereingang**: `rPhys` (physikalischer Wert)

Dieser Baustein empfängt über seinen Adaptereingang `rPhys` den physikalischen Wert und schreibt ihn unverändert auf die Ausgabeadresse `OutputNumber_N3`.

Beide Bausteine werden aus den importierten Bibliotheken `Uebungen::const::UT::DefaultPool_Numeric::InputNumber_I3` und `Uebungen::const::UT::DefaultPool_Numeric::OutputNumber_N3` konfiguriert.

Programmablauf und Verbindungen

Die Verbindung zwischen den beiden Bausteinen erfolgt über eine einzige **Adapterverbindung**:

- Quelle: `NumericValue_PHYSA.rPhys`
- Ziel: `Q_NumericValue_PHYSA.rPhys`

Der Datenfluss ist geradlinig: Sobald `NumericValue_PHYSA` aktiv ist (QI=TRUE), stellt es den aktuellen physikalischen Wert von `I3` bereit. Dieser Wert wird direkt an `Q_NumericValue_PHYSA` weitergeleitet, der ihn auf `N3` ausgibt. Es findet keine Umwandlung oder Verarbeitung statt – ein reines Durchschleifen.

Einordnung: Die Übung ist für Einsteiger geeignet, die die Grundlagen der ISO-BUS-Bausteinverwendung und des Signalflusses in 4diac-IDE erlernen möchten. Sie zeigt die direkte Kopplung von Ein- und Ausgabebausteinen über physikalische Adapter.

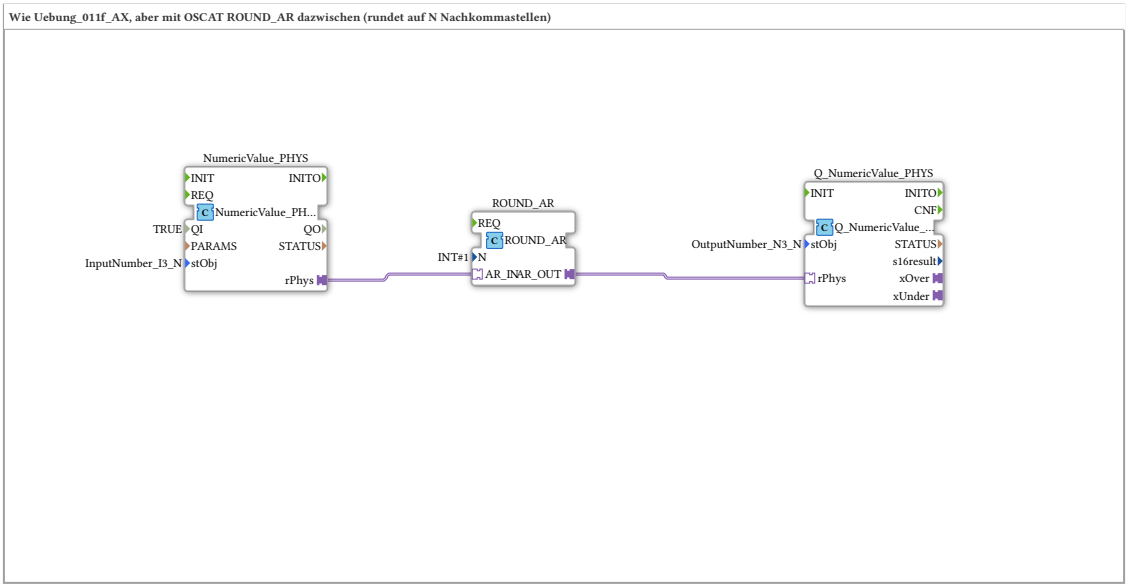
Zusammenfassung

Die Übung **Uebung_011f_PHYSA** realisiert eine einfache Durchschleifung eines physikalischen Werts ohne Konvertierung. Der Baustein `NumericValue_PHYSA` liest den Wert von `InputNumber_I3`, gibt ihn über den Adapterausgang `rPhys` an `Q_NumericValue_PHYSA` weiter, welcher den Wert auf `OutputNumber_N3` ausgibt. Die Übung ist ein grundlegendes Beispiel für die Nutzung von physikalischen Werten und Adapterverbindungen in 4diac.

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

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

2.1.2.182 Uebung_011f_ROUND_AR_AX: Physikalischer Zahlenwert mit OSCAT ROUND_AR runden



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung entspricht `Uebung_011f_AX`, jedoch ist zwischen Eingabe und Ausgabe der OSCAT-Baustein `ROUND_AR` eingefügt, der den Wert auf N Nachkommastellen rundet. Der physikalische Zahlenwert der Eingabe `InputNumber_I3_N` wird als `REAL`-Adapter (`rPhys`) durch `ROUND_AR` geführt und an der Ausgabe `OutputNumber_N3_N` angezeigt.

Verwendete Funktionsbausteine (FBs)

- **NumericValue_PHYS**: Eingabe eines physikalischen Zahlenwerts
  - **Typ**: `isobus::UT::io::NumericValue::NumericValue_PHYSA`
  - **Parameter**: `QI = TRUE`, `stObj = InputNumber_I3_N`
  - **Erklärung**: Liefert den eingegebenen Zahlenwert als physikalischen Wert über den Adapter `rPhys`.
- **ROUND_AR**: Rundet auf N Nachkommastellen
  - **Typ**: `OSCAT_adapter::Mathematical::ROUND_AR`
  - **Parameter**: `N = 1`
  - **Erklärung**: Rundet den Wert von `AR_IN` auf `N` Nachkommastellen (hier eine) und gibt ihn über `AR_OUT` aus.
- **Q_NumericValue_PHYS**: Ausgabe eines physikalischen Zahlenwerts
  - **Typ**: `isobus::UT::Q::Q_NumericValue_PHYSA`
  - **Parameter**: `stObj = OutputNumber_N3_N`
  - **Erklärung**: Zeigt den über `rPhys` empfangenen Wert an `OutputNumber_N3_N` an.

Programmablauf und Verbindungen

1. `NumericValue_PHYS.rPhys` ⟶ `ROUND_AR.AR_IN` (AdapterConnection): Der physikalische Eingabewert geht in den Rundungsbaustein.
2. `ROUND_AR.AR_OUT` ⟶ `Q_NumericValue_PHYS.rPhys` (AdapterConnection): Der gerundete Wert wird an die Ausgabe weitergereicht.

Beispiel (aus dem Kommentar im Netzwerk): Eingabe I3 = −500.00 ⟶ `rPhys` = −500.0 ⟶ `ROUND_AR` (`N = 1`) ⟶ −500.0 ⟶ `Q_NumericValue_PHYS` (N3) ⟶ N3 zeigt −500.00.

Zusammenfassung

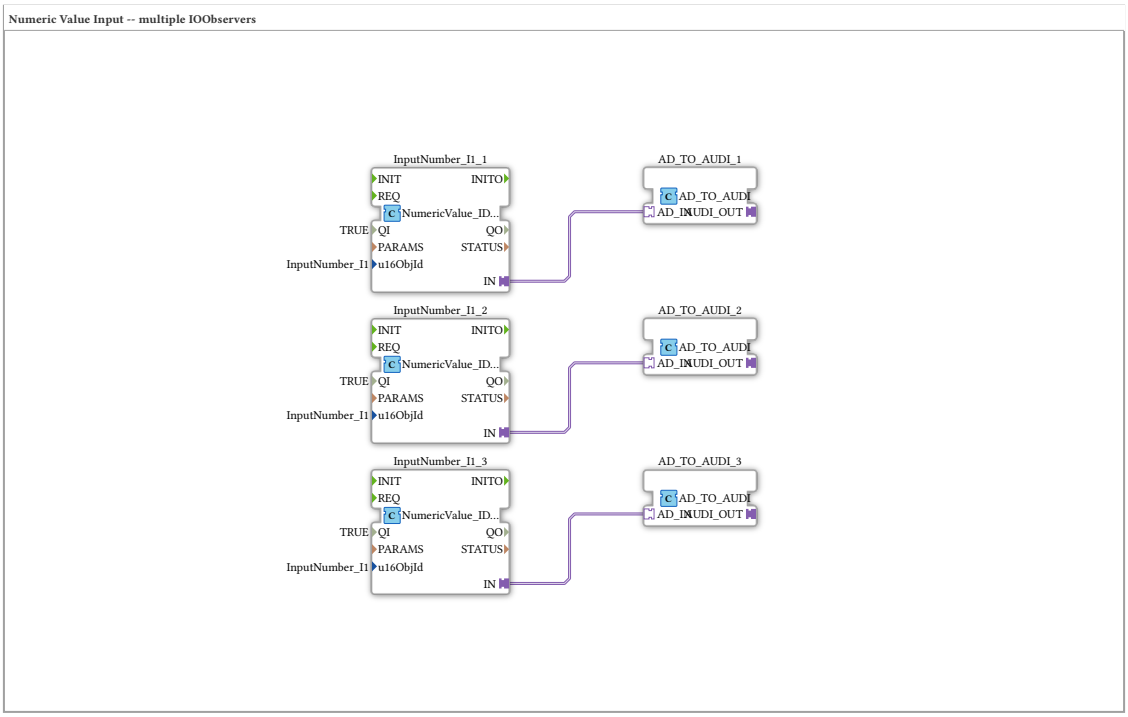
Die Übung zeigt, wie sich in den Adapterpfad eines physikalischen Zahlenwerts ein OSCAT-Baustein (`ROUND_AR`) einschleifen lässt, ohne die Ein- und Ausgabebausteine zu ändern. Gegenüber `Uebung_011f_AX` wird der Wert dadurch auf die eingestellte Zahl von Nachkommastellen gerundet.

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

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

2.1.2.183 Uebung_011g1_AX: Numeric Value Input – multiple IOObservers

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_011g1_AX (Numeric Value Input – multiple IOObservers).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Numeric Value Input – multiple IOObservers**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_011g1_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **InputNumber_I1_1**: Instanz des Typs isobus::UT::io::NumericValue::NumericValue_IDA.
  - Parameter QI = TRUE
  - Parameter u16ObjId = InputNumber_I1
- **InputNumber_I1_2**: Instanz des Typs isobus::UT::io::NumericValue::NumericValue_IDA.
  - Parameter QI = TRUE
  - Parameter u16ObjId = InputNumber_I1
- **InputNumber_I1_3**: Instanz des Typs isobus::UT::io::NumericValue::NumericValue_IDA.
  - Parameter QI = TRUE
  - Parameter u16ObjId = InputNumber_I1
- **AD_TO_AUDI_1**: Instanz des Typs adapter::conversion::unidirectional::AD_TO_AUDI.
- **AD_TO_AUDI_2**: Instanz des Typs adapter::conversion::unidirectional::AD_TO_AUDI.
- **AD_TO_AUDI_3**: Instanz des Typs adapter::conversion::unidirectional::AD_TO_AUDI.

Verbindungen und Schnittstellen

Adapterverbindungen:

- InputNumber_I1_1.IN -> AD_TO_AUDI_1.AD_IN
- InputNumber_I1_2.IN -> AD_TO_AUDI_2.AD_IN

- InputNumber_I1_3.IN -> AD_TO_AUDI_3.AD_IN

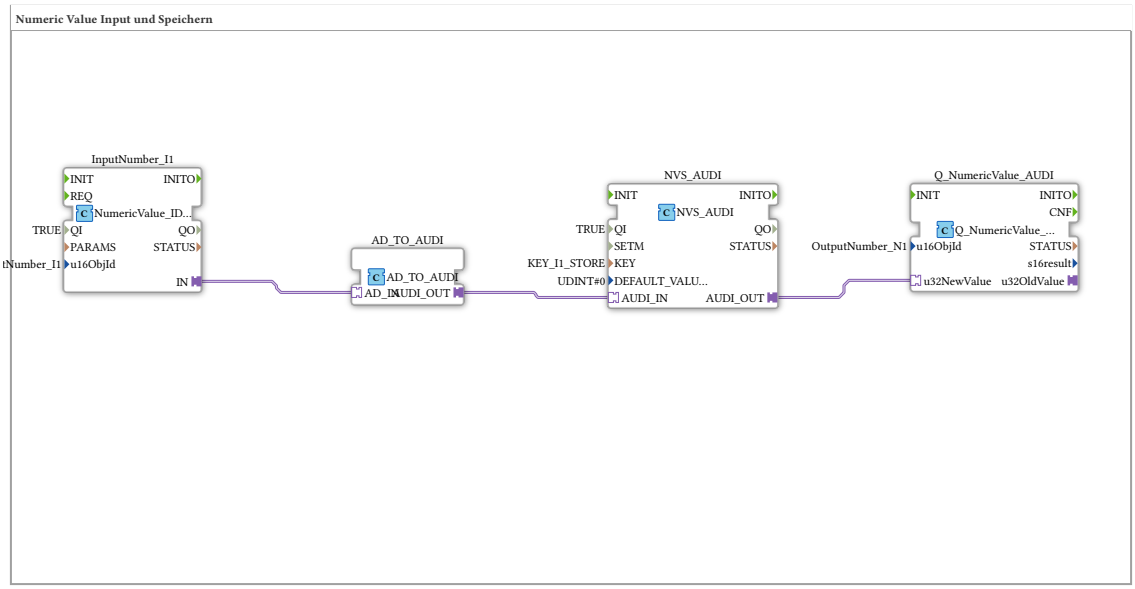
Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_011g1_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.184 Uebung_012_AUDI: Numeric Value Input und Speichern



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert die Aufnahme eines numerischen Werts über den isobus I/O-Stack, die Konvertierung in ein für die Audi-Steuerung kompatibles Format und die persistente Speicherung des Werts im nichtflüchtigen Speicher (NVS). Der gespeicherte Wert wird anschließend über einen weiteren isobus-Ausgabebaustein ausgegeben. Ziel ist das Verständnis des Datenflusses zwischen Eingabe, Konvertierung, Speicherung und Ausgabe.

Verwendete Funktionsbausteine (FBs)

- **AD_TO_AUDI**
  - **Typ**: `adapter::conversion::unidirectional::AD_TO_AUDI`
  - **Parameter**: keine
  - **Funktionsweise**: Wandelt das vom Eingabebaustein kommende Adapter-Signal (isobus-Seite) in ein adapter-basiertes Signal für die Audi-Komponenten um. Stellt die unidirektionale Verbindung zwischen dem isobus-Adapter und dem NVS-Adapter her.
- **InputNumber_I1**
  - **Typ**: `isobus::UT::io::NumericValue::NumericValue_IDA`
  - **Parameter**:
    - `QI` = `TRUE`
    - `u16ObjId` = `InputNumber_I1`

  - **Funktionsweise**: Empfängt einen numerischen Wert aus dem isobus-Netzwerk (definiert durch die Objekt-ID `InputNumber_I1`) und gibt diesen über seinen Adapter-Ausgang (`IN`) an den Konvertierungsbaustein weiter.

- **NVS_AUDI**
  - **Typ**: `logiBUS::storage::esp32_nvs::NVS_AUDI`
  - **Parameter**:
    - `QI` = `TRUE`
    - `KEY` = `KEY_I1_STORE`
    - `DEFAULT_VALUE` = `UDINT#0`
  - **Funktionsweise**: Speichert den an seinem Adapter-Eingang anliegenden Wert persistent im nichtflüchtigen Speicher unter dem Schlüssel `KEY_I1_STORE`. Falls noch kein Wert gespeichert ist, wird der Default-Wert `0` verwendet. Der gespeicherte Wert wird über den Adapter-Ausgang bereitgestellt.

- **Q_NumericValue_AUDI**
  - **Typ**: `isobus::UT::Q::Q_NumericValue_AUDI`
  - **Parameter**:
    - `u16ObjId` = `OutputNumber_N1`
  - **Funktionsweise**: Nimmt den vom NVS-Baustein gelieferten numerischen Wert entgegen und gibt ihn unter der isobus-Objekt-ID `OutputNumber_N1` im Netzwerk aus.

Programmablauf und Verbindungen

Die Bausteine sind ausschließlich über Adapter-Verbindungen verknüpft:

1. Der **InputNumber_I1**-Baustein liest einen numerischen Wert aus dem isobus-Netzwerk und sendet ihn über seinen Adapter-Ausgang `IN` an den **AD_TO_AUDI**-Baustein.
2. **AD_TO_AUDI** konvertiert das eingehende isobus-Adapter-Signal in ein Audi-kompatibles Adapter-Signal und leitet es über seinen Ausgang `AUDI_OUT` an den **NVS_AUDI**-Baustein weiter.
3. **NVS_AUDI** speichert den empfangenen Wert persistent und stellt ihn gleichzeitig über seinen Adapter-Ausgang `AUDI_OUT` bereit.
4. Der Ausgang von **NVS_AUDI** ist mit dem Daten-Eingang (`u32NewValue`) des **Q_NumericValue_AUDI**-Bausteins verbunden, der den Wert schließlich auf dem isobus-Ausgang `OutputNumber_N1` veröffentlicht.

Die gesamte Verarbeitung erfolgt ereignisgesteuert: Sobald ein neuer Wert am Eingang ankommt, durchläuft er die Kette und wird sowohl gespeichert als auch direkt wieder ausgegeben.

Zusammenfassung

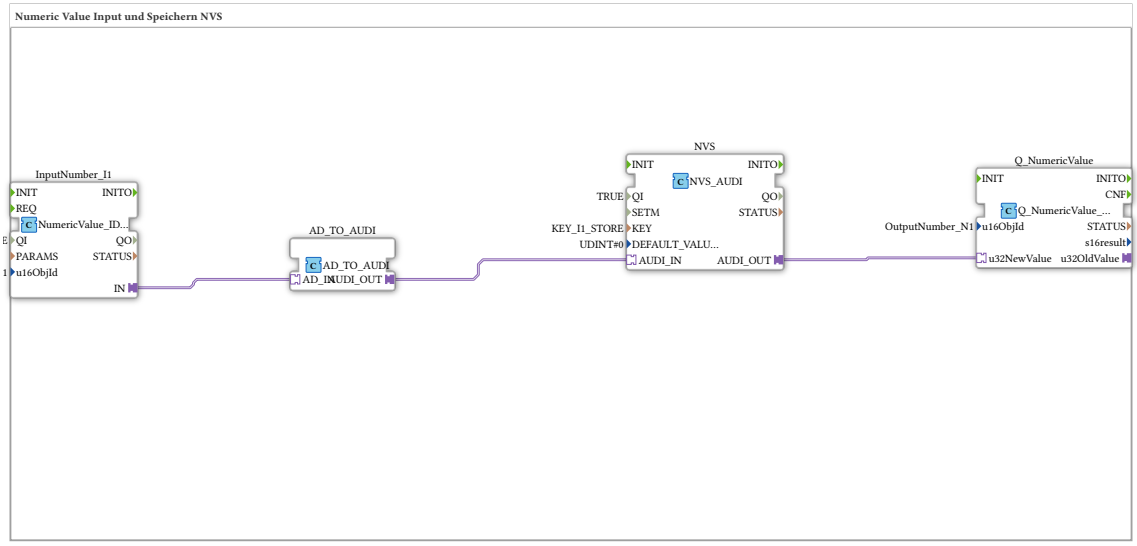
In dieser Übung wird der gesamte Pfad eines numerischen Werts von der isobus-Eingabe über eine Adapter-Konvertierung und persistente Speicherung bis hin zur Netzwerkausgabe abgebildet. Die Verwendung der Adapter-Schnittstellen ermöglicht eine lose Kopplung zwischen den Komponenten. Der Lernende erhält einen praktischen Einblick in die Datenverarbeitung mit isobus-Bausteinen und die Integration eines nichtflüchtigen Speichers in eine 4diac-Applikation.

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

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

2.1.2.185 Uebung_012_AX: Numeric Value Input und Speichern NVS

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_012_AX (Numeric Value Input und Speichern NVS).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Numeric Value Input und Speichern NVS**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_012_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **InputNumber_I1**: Instanz des Typs isobus::UT::io::NumericValue::NumericValue_IDA.
  - Parameter QI = TRUE
  - Parameter u16ObjId = InputNumber_I1
- **AD_TO_AUDI**: Instanz des Typs adapter::conversion::unidirectional::AD_TO_AUDI.
- **NVS**: Instanz des Typs logiBUS::storage::esp32_nvs::NVS_AUDI.
  - Parameter QI = TRUE
  - Parameter KEY = KEY_I1_STORE
  - Parameter DEFAULT_VALUE = UDINT#0
- **Q_NumericValue**: Instanz des Typs isobus::UT::Q::Q_NumericValue_AUDI.
  - Parameter u16ObjId = OutputNumber_N1

Verbindungen und Schnittstellen

Adapterverbindungen:

- InputNumber_I1.IN -> AD_TO_AUDI.AD_IN
- AD_TO_AUDI.AUDI_OUT -> NVS.AUDI_IN
- NVS.AUDI_OUT -> Q_NumericValue.u32NewValue

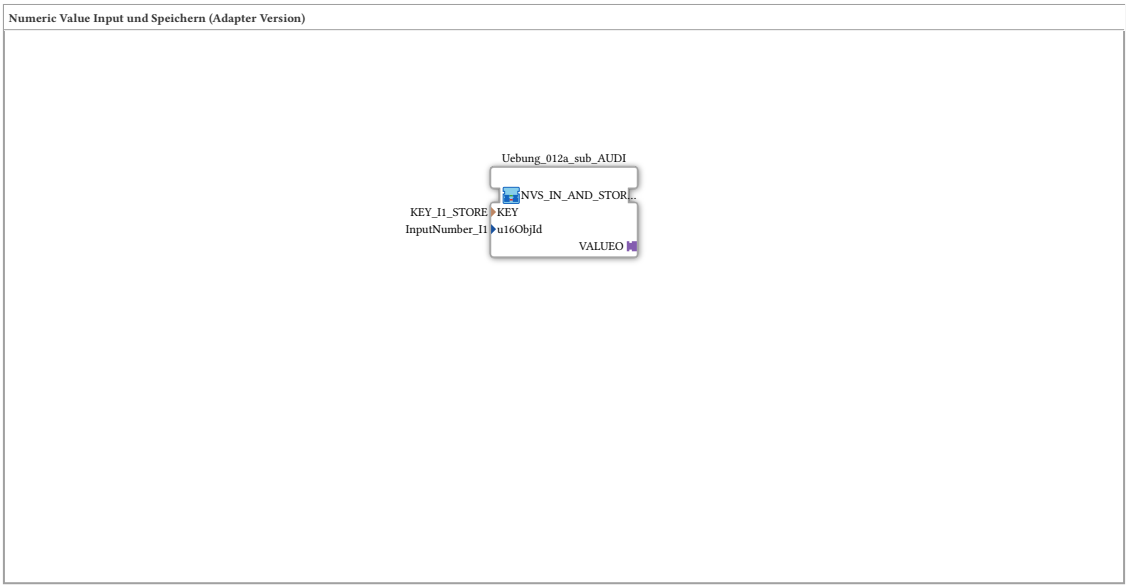
Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_012_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.186 Uebung_012a_AX: Numeric Value Input und Speichern (Adapter Version)



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung vermittelt den Umgang mit Adaptern zur numerischen Werteingabe und deren persistenter Speicherung im nichtflüchtigen Speicher (NVS). Der Funktionsbaustein `Uebung_012a_AX` kapselt die gesamte Logik in einem SubApp, der über einen Adapter einen Zahlenwert empfängt und unter einem definierten Schlüssel speichert. Die Adapter-Variante ermöglicht eine flexible Anbindung an verschiedene Eingabequellen.

Der Baustein selbst besitzt keine sichtbaren Ein-/Ausgangsschnittstellen – die Kommunikation erfolgt vollständig über den Adapter und die konfigurierten Konstanten.

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: Uebung_012a_sub_AUDI

- **Typ**: `MyLib::sys::NVS_IN_AND_STORE_AUDI`
- **Verwendete interne FBs**: Nicht näher spezifiziert – der Baustein enthält eine gekapselte Implementierung zur Adapter-basierten Eingabe und NVS-Speicherung.
- **Parameter**:
  - `KEY = KEY_I1_STORE` (Schlüssel, unter dem der Wert im NVS abgelegt wird; definiert in `Uebungen::const::NVS::NVS_Keys`)
  - `u16ObjId = InputNumber_I1` (Identifikation des Eingabewerts; definiert in `Uebungen::const::UT::DefaultPool`)
- **Funktionsweise**: Der FB liest bei Aktivierung den aktuellen numerischen Wert von einem Adapter ein und speichert diesen persistent im NVS unter dem angegebenen `KEY`. Die Objekt-ID `InputNumber_I1` referenziert die Bereitstellung des Werts durch die übergeordnete Umgebung (z. B. eine HMI-Eingabe oder eine übergeordnete Steuerung).

Programmablauf und Verbindungen

Der SubApp `Uebung_012a_AX` besteht aus einem einzelnen Netzwerk, das den FB `Uebung_012a_sub_AUDI` enthält. Es sind keine weiteren Verbindungen oder Ereignisverkettungen sichtbar, da alle Interaktionen über den Adapter und die parametrierten Konstanten gesteuert werden.

Ablauf:

1. Die übergeordnete Applikation aktiviert den SubApp (z. B. durch ein Ereignis).
2. Der FB `Uebung_012a_sub_AUDI` liest den aktuellen Wert vom Adapter ein.
3. Der eingelesene Wert wird unter dem Schlüssel `KEY_I1_STORE` im NVS gespeichert.

Lernziele:

- Verständnis der Adapter-Schnittstelle für Eingabewerte.
- Kennenlernen der persistenten Speicherung mit NVS.
- Umgang mit konstanten Schlüsseln und Objekt-IDs aus getrennten Const-Quellen.

Schwierigkeitsgrad: Mittel **Vorkenntnisse**: Grundlagen der 4diac-IDE, Umgang mit SubApps und Konstanten, Adapter-Konzept.

Hinweise zur Einbindung:

- Die Konstanten `KEY_I1_STORE` und `InputNumber_I1` müssen in den entsprechenden `const`-Ressourcen des Projekts definiert sein.
- Der Adapter muss von der aufrufenden Umgebung bereitgestellt werden (z. B. über eine HMI-Komponente).

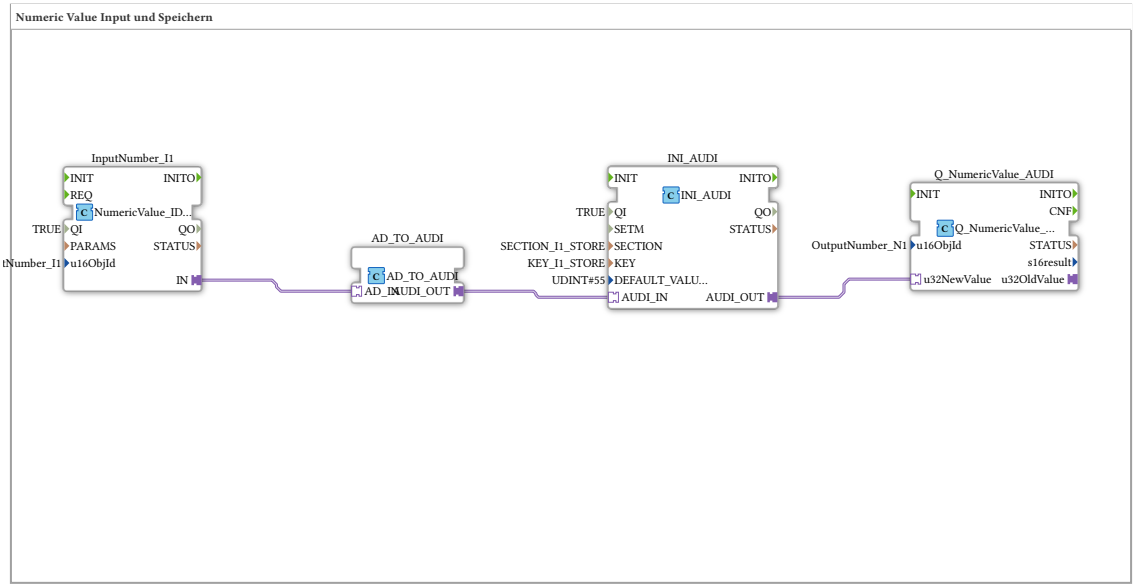
Zusammenfassung

Die Übung `Uebung_012a_AX` demonstriert die effiziente Nutzung von Adaptern zur numerischen Werteingabe und deren persistente Ablage im NVS. Durch die Kapselung in einem SubApp und die Verwendung von Konstanten wird eine hohe Wiederverwendbarkeit und Konfigurierbarkeit erreicht. Der Baustein eignet sich besonders für Szenarien, in denen flexible Eingabequellen (z. B. Tastatur, Busteilsysteme) an eine einheitliche Speicherlogik angebunden werden sollen.

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

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

2.1.2.187 Uebung_012b_AUDI: Numeric Value Input und Speichern



• — ○ • — ○ • — ○ •

Einleitung

Diese Übung demonstriert die Aufnahme eines numerischen Werts über einen ISOBUS-Eingang, dessen Umwandlung in ein speicherbares Format und die persistente Speicherung mittels eines INI-basierten Speichermechanismus. Der gespeicherte Wert wird anschließend wieder ausgelesen und als ISOBUS-Ausgabewert bereitgestellt. Die Funktionsbausteine kommunizieren über Adapter-Schnittstellen (AUDI), die eine standardisierte Datenübergabe ermöglichen.

Verwendete Funktionsbausteine (FBs)

Die Übung besteht aus einem Subapplikationsnetzwerk, das vier Funktionsbausteine sowie deren Adapterverbindungen enthält. Es werden keine weiteren Sub-Bausteine (verschachtelte SubApplikationen) verwendet.

- **InputNumber_I1** (Typ: `isobus::UT::io::NumericValue::NumericValue_IDA`) – Parameter: `QI = TRUE`, `u16ObjId = InputNumber_I1` – Eingang: Adapter-Schnittstelle `IN` – Funktion: Liest einen numerischen Wert von einem ISOBUS-Eingang (Object-ID `InputNumber_I1`). Der ausgegebene Wert wird über den Adapter-Ausgang `AD_IN` an die nächste Stufe weitergegeben.
- **AD_TO_AUDI** (Typ: `adapter::conversion::unidirectional::AD_TO_AUDI`) – Funktion: Wandelt das Datenformat des vorherigen Adapters (`AD`-Schnittstelle) in das AUDI-Format um. Dadurch wird die Kompatibilität zwischen den unterschiedlichen Adaptertypen hergestellt.
- **INI_AUDI** (Typ: `eclipse4diac::storage::INI_AUDI`) – Parameter: `QI = TRUE`, `SECTION = SECTION_I1_STORE`, `KEY = KEY_I1_STORE`, `DEFAULT_VALUE = UDINT#55` – Eingang: Adapter-Schnittstelle `AUDI_IN` – Funktion: Schreibt den übergebenen numerischen Wert in einen INI-Speicher (Abschnitt `SECTION_I1_STORE`, Schlüssel `KEY_I1_STORE`). Wird kein gültiger Wert geliefert, wird der Defaultwert 55 verwendet. Der gespeicherte Wert wird über den Adapter-Ausgang `AUDI_OUT` ausgegeben.
- **Q_NumericValue_AUDI** (Typ: `isobus::UT::Q::Q_NumericValue_AUDI`) – Parameter: `u16ObjId = OutputNumber_N1` – Eingang: Datenanschluss `u32NewValue` (verbunden mit dem Adapter-Ausgang von `INI_AUDI`) – Funktion: Setzt den übergebenen 32-Bit-Wert als neuen Ausgabewert für die ISOBUS-Objekt-ID `OutputNumber_N1`. Der Wert wird so auf dem ISOBUS-Datenfeld ausgegeben.

Compiler-Importe

Die Übung importiert folgende Konstanten aus der Bibliothek `Uebungen::const::NVS::NVS_Keys` und `Uebungen::const::UT::DefaultPool`:

- `KEY_I1_STORE` – der Schlüssel für den INI-Speicher
- `SECTION_I1_STORE` – die Abschnittskennung für den INI-Speicher
- `InputNumber_I1` – die ISOBUS-Objekt-ID des Eingabewerts
- `OutputNumber_N1` – die ISOBUS-Objekt-ID des Ausgabewerts

Programmablauf und Verbindungen

1. Der Baustein **InputNumber_I1** empfängt einen numerischen Wert von der ISOBUS-Schnittstelle und gibt diesen über seinen Adapter-Ausgang `IN` aus.
2. Dieser Adapter-Ausgang ist verbunden mit dem Adapter-Eingang `AD_IN` von **AD_TO_AUDI**. Dieser Baustein konvertiert das Datenformat und stellt den Wert an seinem Ausgang `AUDI_OUT` bereit.
3. Der Ausgang `AD_TO_AUDI.AUDI_OUT` ist verbunden mit dem Adapter-Eingang `AUDI_IN` von **INI_AUDI**. Dieser speichert den Wert persistent in einem INI-Abschnitt.
4. Der gespeicherte Wert wird vom Adapter-Ausgang `INI_AUDI.AUDI_OUT` an den Datenanschluss `u32NewValue` von **Q_NumericValue_AUDI** übergeben.
5. **Q_NumericValue_AUDI** setzt daraufhin den ISOBUS-Ausgabewert mit der Objekt-ID `OutputNumber_N1` auf diesen Wert.

Die gesamte Datenkette ist unidirektional und arbeitet ohne explizite Ereignissteuerung – die Ausführung erfolgt zyklisch durch die Laufzeitumgebung.

Lernziele:

- Verständnis der Adapter-basierte Datenübergabe zwischen unterschiedlichen Funktionsbausteinen.
- Kennenlernen des INI-Speicher-Bausteins (`INI_AUDI`) zur persistenten Speicherung von Werten.
- Anwendung von ISOBUS-Ein-/Ausgabe-Bausteinen mit konfigurierbaren Objekt-IDs.

Vorkenntnisse: Grundlagen der 4diac-IDE, Erstellung von Subapplikationen, Umgang mit Adaptern.

Hinweise zur Übung: Die Konstanten `SECTION_I1_STORE` und `KEY_I1_STORE` müssen im Projekt als NVS-Konstanten definiert sein. Der Defaultwert von 55 dient als Fallback, falls noch kein Wert gespeichert wurde.

Zusammenfassung

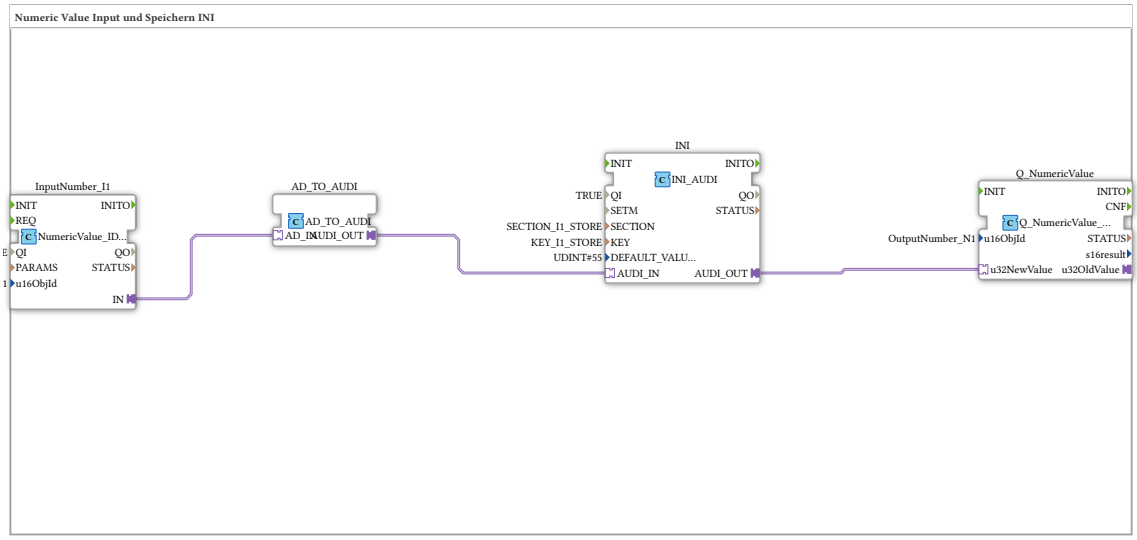
Die Übung **Uebung_012b_AUDI** zeigt einen vollständigen Datenpfad von der ISOBUS-Eingabe über eine Formatkonvertierung, persistente Speicherung in einer INI-Struktur bis zur ISOBUS-Ausgabe. Sie veranschaulicht den Einsatz von Adaptern zur Kopplung unterschiedlicher Bausteintypen und die Verwendung von Speicherbausteinen für dauerhafte Datenhaltung. Nach erfolgreicher Durchführung können die Teilnehmer selbstständig ähnliche Datenspeicher- und Weiterleitungsketten in eigenen Projekten umsetzen.

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

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

2.1.2.188 Uebung_012b_AX: Numeric Value Input und Speichern INI

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_012b_AX (Numeric Value Input und Speichern INI).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Numeric Value Input und Speichern INI**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_012b_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **InputNumber_I1**: Instanz des Typs isobus::UT::io::NumericValue::NumericValue_IDA.
  - Parameter QI = TRUE
  - Parameter u16ObjId = InputNumber_I1
- **AD_TO_AUDI**: Instanz des Typs adapter::conversion::unidirectional::AD_TO_AUDI.
- **INI**: Instanz des Typs eclipse4diac::storage::INI_AUDI.
  - Parameter QI = TRUE
  - Parameter SECTION = SECTION_I1_STORE
  - Parameter KEY = KEY_I1_STORE
  - Parameter DEFAULT_VALUE = UDINT#55
- **Q_NumericValue**: Instanz des Typs isobus::UT::Q::Q_NumericValue_AUDI.
  - Parameter u16ObjId = OutputNumber_N1

Verbindungen und Schnittstellen

Adapterverbindungen:

- InputNumber_I1.IN -> AD_TO_AUDI.AD_IN
- AD_TO_AUDI.AUDI_OUT -> INI.AUDI_IN
- INI.AUDI_OUT -> Q_NumericValue.u32NewValue

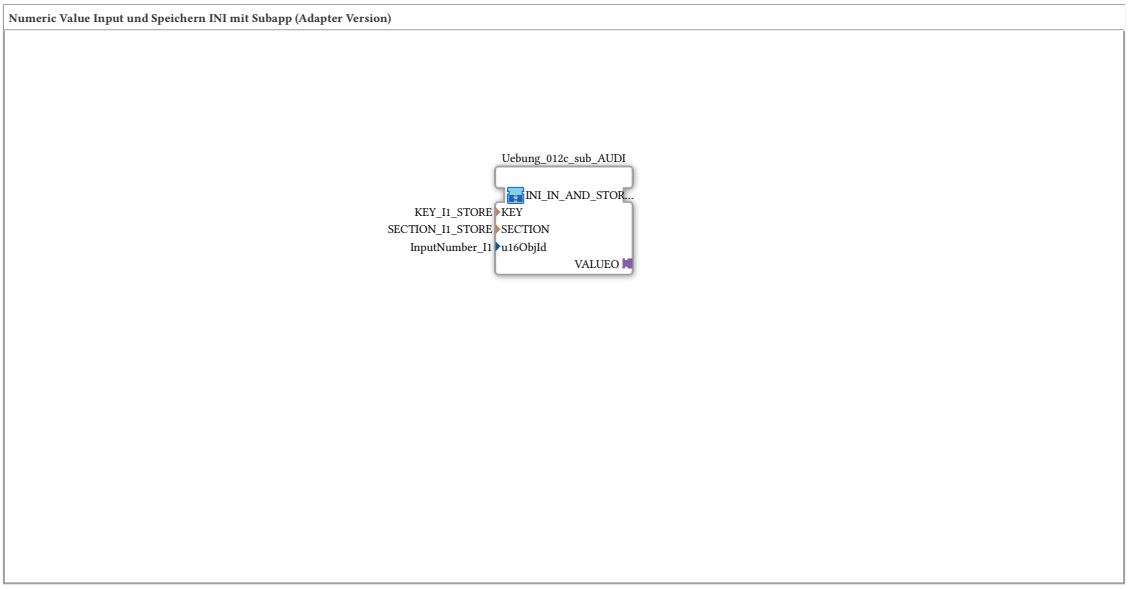
Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.

3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung
Die Übung Uebung_012b_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.189 Uebung_012c_AUDI: Numeric Value Input und Speichern INI mit Subapp (Adapter Version)



• – ◦ • – ◦ • – ◦ •

Einleitung
Diese Übung demonstriert die Realisierung einer numerischen Werteingabe mit anschließender Speicherung in einem nichtflüchtigen Speicher (NVS) unter Verwendung einer SubApp. Das Besondere ist die Nutzung einer SubApp in der **Adapter-Version**, welche es ermöglicht, die Ein- und Ausgänge über einen Adapter-Mechanismus zu kapseln. Die SubApp **Uebung_012c_AUDI** enthält einen vorgefertigten Sub-Baustein, der den gesamten Prozess der Eingabe und Speicherung übernimmt.

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: Uebung_012c_sub_AUDI
- **Typ**: `MyLib::sys::INI_IN_AND_STORE_AUDI`
- **Verwendete interne FBs**: Die internen FBs dieses Bausteins sind nicht im Detail einsehbar, da es sich um einen vordefinierten Baustein aus der Bibliothek `MyLib::sys` handelt. Er besteht vermutlich aus einer Kette von Bausteinen für die Wertaufnahme, Konvertierung und die NVS-Speicherung.
- **Funktionsweise**: Der Baustein liest einen numerischen Wert aus dem `DefaultPool` unter der Objekt-ID `InputNumber_I1` (importiert aus `Uebungen::const::UT::DefaultPool`). Diesen Wert speichert er im nichtflüchtigen Speicher (NVS) unter dem Schlüssel `KEY_I1_STORE` in der Sektion `SECTION_I1_STORE` (beide importiert aus `Uebungen::const::NVS::NVS_Keys`). Die Parameter `KEY`, `SECTION` und `u16ObjId` werden der SubApp beim Einfügen übergeben.

Parameter des Sub-Bausteins

Parameter	Wert	Quelle / Bedeutung
`KEY`	`KEY_I1_STORE`	Konstante: Schlüssel für den NVS-Eintrag
`SECTION`	`SECTION_I1_STORE`	Konstante: Sektion in der NVS-Struktur
`u16ObjId`	`InputNumber_I1`	Objekt-ID aus dem DefaultPool, die den Eingabewert liefert

Programmablauf und Verbindungen

Die Haupt-SubApp `Uebung_012c_AUDI` besitzt **keine eigenen Ein-/Ausgangsschnittstellen** (`SubAppInterfaceList` ist leer). Der gesamte Funktionsumfang wird durch den eingebetteten Sub-Baustein `Uebung_012c_sub_AUDI` abgedeckt.

- **Ablauf**: Sobald die SubApp aktiviert wird (z. B. durch einen übergeordneten Baustein), startet der Sub-Baustein `INI_IN_AND_STORE_AUDI` seinen internen Ablauf. Er holt den aktuellen Wert der Spannweite `InputNumber_I1` aus dem `DefaultPool` und speichert diesen persistent im NVS unter den angegebenen Schlüssel- und Sektionskonstanten.
- **Verbindungen**: Es sind keine expliziten Daten- oder Ereignisverbindungen innerhalb der SubApp sichtbar, da der Sub-Baustein seine Funktionalität vollständig intern kapselt. Die SubApp dient als Wrapper, um die Konfiguration (über Parameter) zu ermöglichen und die Wiederverwendung zu vereinfachen.
- **Lernziele**: Verständnis für die Verwendung von SubApps in 4diac, insbesondere als Adapter-Version. Kennenlernen der NVS-Speicherung sowie der Verknüpfung von Eingabewerten aus einem DefaultPool mit persistenten Speicherorten.
- **Schwierigkeitsgrad**: Mittel – Grundlegende Kenntnisse in 4diac und der Funktionsblock-Modellierung werden vorausgesetzt.

Zusammenfassung

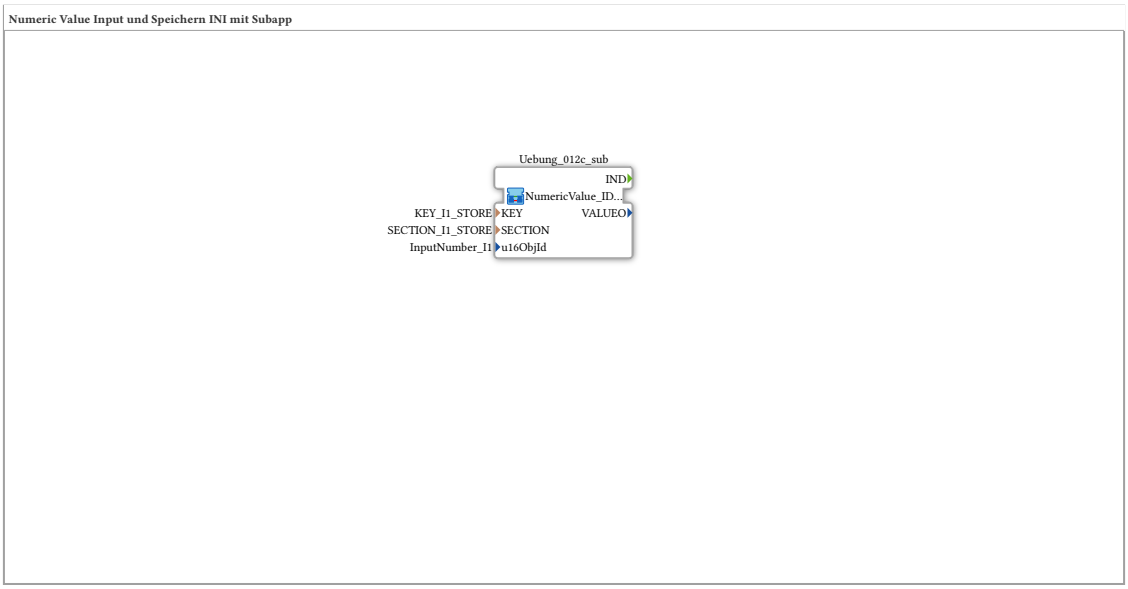
Die Übung *Uebung_012c_AUDI* zeigt, wie ein numerischer Wert über eine vordefinierte SubApp (Adapter-Version) eingelesen und in einem nichtflüchtigen Speicher abgelegt wird. Dabei kommen Konstanten für Schlüssel und Sektion sowie eine Objekt-ID aus dem DefaultPool zum Einsatz. Der Lernende erhält Einblick in die Kapselung von Funktionslogik durch SubApps und in die Handhabung der NVS-Bibliothek in 4diac.

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

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

2.1.2.190 Uebung_012c_AX: Numeric Value Input und Speichern INI mit Subapp

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_012c_AX (Numeric Value Input und Speichern INI mit Subapp).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Numeric Value Input und Speichern INI mit Subapp**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_012c_AX.SUB, welche die folgende Bausteinstruktur verwendet:

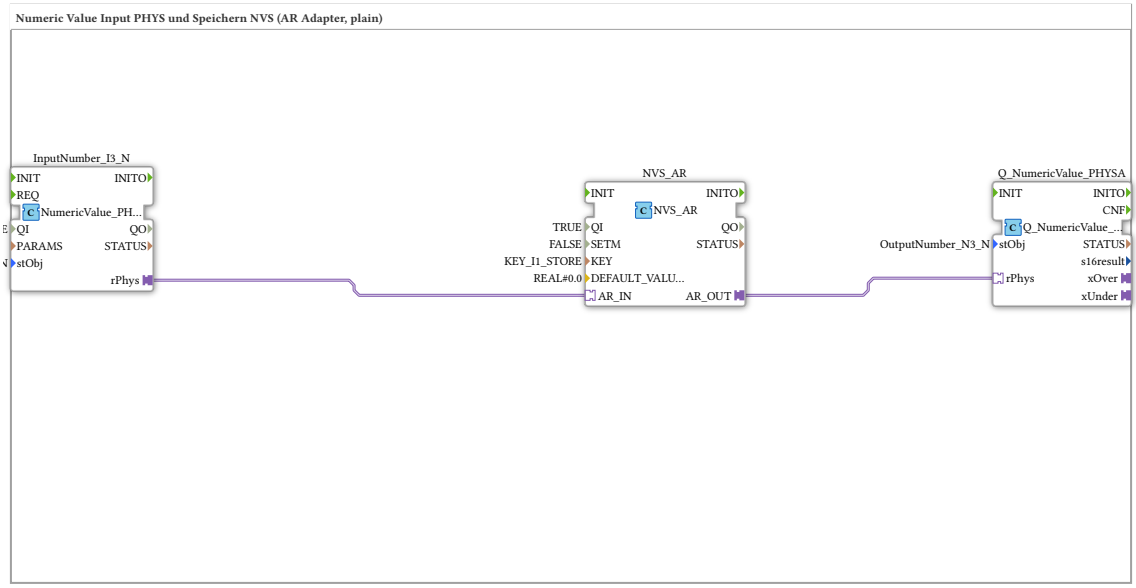
Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Signalweiterleitung**: Die Bausteine verarbeiten Daten- und Steuersignale gemäß ihrer Konfiguration.

Zusammenfassung

Die Übung Uebung_012c_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.191 Uebung_012d_AR: Numeric Value Input PHYS und Speichern NVS (AR Adapter, plain)



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung demonstriert die Erfassung eines numerischen Werts über einen physikalischen Eingang (PHYS), das Speichern des Werts im nichtflüchtigen Speicher (NVS) sowie die anschließende Ausgabe. Die Kommunikation zwischen den Funktionsbausteinen erfolgt über einen AR-Adapter (Adapter-Interface), ohne Verwendung von Unterbausteinen.

Verwendete Funktionsbausteine (FBs)

- **InputNumber_I3**
  - **Typ**: `isobus::UT::io::NumericValue::NumericValue_PHYSA`
  - **Parameter**:
    - `QI` = `TRUE` (Aktivierung des Bausteins)
    - `stObj` = `InputNumber_I3` (Referenz auf das physische Eingangsobjekt)
  - **Funktion**: Liest einen numerischen Wert von einer physischen Eingangsschnittstelle und stellt ihn über den Ausgangsadapter `rPhys` bereit.
- **NVS_AR**
  - **Typ**: `logiBUS::storage::esp32_nvs::NVS_AR`
  - **Parameter**:
    - `QI` = `TRUE` (Aktivierung)
    - `KEY` = `KEY_I1_STORE` (Speicher-Schlüssel im NVS)
    - `DEFAULT_VALUE` = `REAL#0.0` (Standardwert, falls noch kein Wert gespeichert ist)

  - **Funktion**: Speichert einen eingehenden Wert im nichtflüchtigen Speicher (NVS) und gibt den gespeicherten (oder standardmäßigen) Wert über den Ausgangsadapter `AR_OUT` weiter. Der Adaptereingang `AR_IN` nimmt Eingabedaten entgegen.

- **Q_NumericValue**
  - **Typ**: `isobus::UT::Q::Q_NumericValue_PHYSA`
  - **Parameter**:
    - `stObj` = `OutputNumber_N3` (Referenz auf das physische Ausgangsobjekt)
  - **Funktion**: Gibt einen empfangenen numerischen Wert über eine physische Ausgangsschnittstelle aus. Die Daten werden über den Adaptereingang `rPhys` bereitgestellt.

Programmablauf und Verbindungen

Die Funktionsbausteine sind über Adapter-Schnittstellen verbunden:

1. **Eingabe**: Der Baustein `InputNumber_I3` liest den aktuellen Wert einer physischen numerischen Eingabe (z. B. Potentiometer oder Sensor) und stellt diesen am Ausgangsadapter `rPhys` bereit.
2. **Speicherung und Weitergabe**: Der Adapterausgang `InputNumber_I3.rPhys` ist mit dem Adaptereingang `NVS_AR.AR_IN` verbunden. Der Baustein `NVS_AR` speichert den empfangenen Wert unter dem Schlüssel `KEY_I1_STORE` im nichtflüchtigen Speicher und gibt den gespeicherten (oder falls nicht vorhanden den Standardwert) über den Ausgangsadapter `AR_OUT` aus.
3. **Ausgabe**: Der Adapterausgang `NVS_AR.AR_OUT` ist mit dem Adaptereingang `Q_NumericValue.rPhys` verbunden. Der Baustein `Q_NumericValue` gibt den erhaltenen Wert auf dem physischen Ausgang `OutputNumber_N3` aus (z. B. Anzeige oder analoges Signal).

Lernziele der Übung:

- Verwendung von AR-Adaptern zur Datenübertragung zwischen Funktionsbausteinen.
- Kombination von physischer Ein-/Ausgabe mit nichtflüchtiger Speicherung.
- Parametrierung von Speicherbausteinen (NVS) mit Schlüsseln und Standardwerten.

Schwierigkeitsgrad: Mittel **Vorkenntnisse:** Grundlegende Kenntnisse der 4diac-IDE, Verständnis von Funktionsbausteinen und Adaptern.

Zusammenfassung

Die Übung `Uebung_012d_AR` realisiert eine einfache Pipeline: physikalischer Eingang ⟶ Speicherung im NVS ⟶ physikalischer Ausgang. Die Datenweitergabe erfolgt ausschließlich über AR-Adapter, sodass keine komplexen Verbindungen zwischen einzelnen Ein-/Ausgängen nötig sind. Der gespeicherte Wert bleibt auch nach einem Neustart erhalten. Die Übung vermittelt den Umgang mit NVS-Speicher und adapterbasierter Kommunikation in der 4diac-Entwicklungsumgebung.

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

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

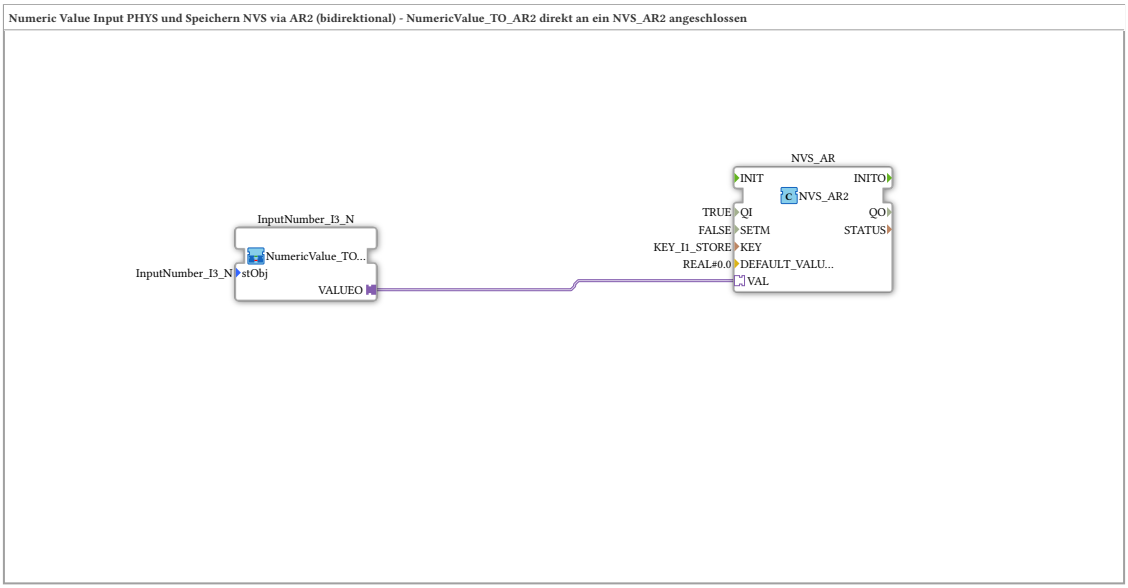
2.1.2.192 Uebung_012d_AR2: Numeric Value Input PHYS und Speichern NVS via AR2 (bidirektional, direkt)

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_012d_AR2`. Teil einer Serie von Übungen (`Uebung_012d_AR2` bis `Uebung_012o_AR2`), die verschiedene Kombinationen aus Eingabequelle (VT-Feld, OPC-UA oder beides) und Speicherziel (NVS-Flash oder INI-Datei) über den bidirektionalen `AR2`-Adapter demonstrieren.

Ziel der Übung

Einen numerischen Wert aus einem VT-Zahlenfeld (`NumericValue_TO_AR2`) entgegennehmen und dauerhaft in NVS (ESP32-Flash) speichern - unter Verwendung des bidirektionalen AR2-Adapters (direkt verdrahtet, ohne Subapp-Kapselung).

Beschreibung und Komponenten



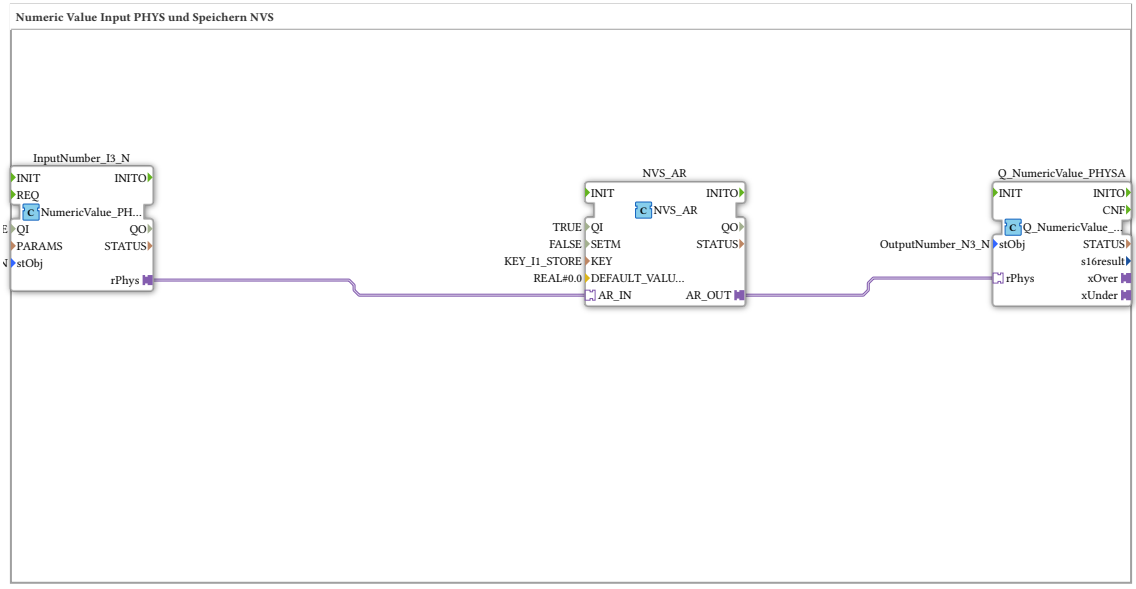
- **`NVS_AR`**: `logiBUS::storage::esp32_nvs::NVS_AR2`, bidirektionaler AR2-Adapter-Baustein, der den Wert direkt persistiert.

Funktionsweise

Der Zahlenwert wird aus einem VT-Zahlenfeld (`NumericValue_TO_AR2`) direkt über den bidirektionalen `AR2`-Adapter an `NVS_AR` weitergegeben, der ihn in NVS (ESP32-Flash) speichert - ohne Zwischenschritt über eine separate Subapp.

2.1.2.193 Uebung_012d_AX: Numeric Value Input PHYS und Speichern NVS

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_012d_AX (Numeric Value Input PHYS und Speichern NVS).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Numeric Value Input PHYS und Speichern NVS**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_012d_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **InputNumber_I3_N**: Instanz des Typs isobus::UT::io::NumericValue::NumericValue_PHYSA.
  - Parameter QI = TRUE
  - Parameter stObj = InputNumber_I3_N
- **NVS_AR**: Instanz des Typs logiBUS::storage::esp32_nvs::NVS_AR.
  - Parameter QI = TRUE
  - Parameter SETM = FALSE
  - Parameter KEY = KEY_I1_STORE
  - Parameter DEFAULT_VALUE = REAL#0.0
- **Q_NumericValue_PHYSA**: Instanz des Typs isobus::UT::Q::Q_NumericValue_PHYSA.
  - Parameter stObj = OutputNumber_N3_N

Verbindungen und Schnittstellen

Adapterverbindungen:

- NVS_AR.AR_OUT -> Q_NumericValue_PHYSA.rPhys
- InputNumber_I3_N.rPhys -> NVS_AR.AR_IN

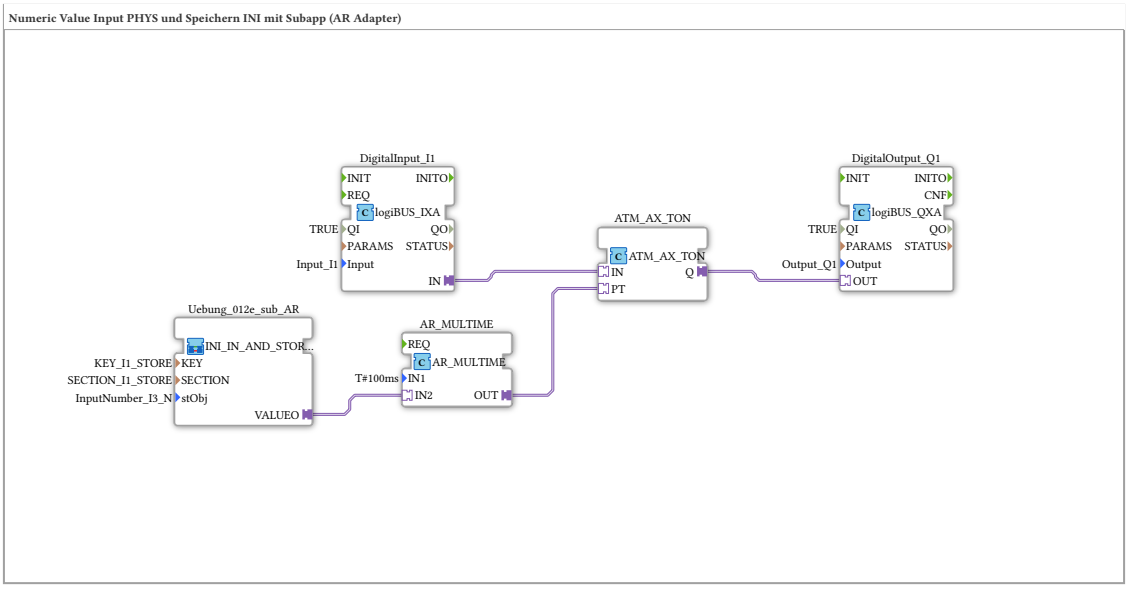
Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_012d_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.194 Uebung_012e2_AR: Numeric Value Input PHYS und Speichern INI mit Subapp (AR Adapter)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert die Verwendung eines **AR-Adapters** zur Kombination eines logiBUS-Digital-Eingangs, einer parametrierbaren Zeitbasis und eines über einen Sub-App-Baustein konfigurierten Timerwerts, der aus einem NVS (Non-Volatile Storage) geladen wird. Ein digitales Eingangssignal startet einen Timer, dessen Ablaufzeit über einen gespeicherten numerischen Wert (INI) eingestellt wird. Der Timerausgang schaltet einen digitalen Ausgang. Die Besonderheit liegt in der AR-Adapter-Verknüpfung zwischen dem Speicherbaustein, der arithmetischen Einheit und dem Timer.

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: Uebung_012e_sub_AR

- **Typ**: `MyLib::sys::INI_IN_AND_STORE_AR`
- **Verwendete interne FBs**: (nicht im bereitgestellten XML enthalten; der Baustein wird als vordefinierter Sub-App-Typ importiert)
- Parameter:
  - `KEY` = `KEY_I1_STORE` (Konstante aus `Uebungen::const::NVS::NVS_Keys`)
  - `SECTION` = `SECTION_I1_STORE` (Konstante aus `Uebungen::const::NVS::NVS_Keys`)
  - `stObj` = `InputNumber_I3` (Konstante aus `Uebungen::const::UT::DefaultPool_Numeric`)
- **Funktionsweise**: Der Sub-Baustein lädt bei Initialisierung einen numerischen Wert (z. B. einen Timer-Sollwert) aus dem NVS unter dem angegebenen Schlüssel und Sektionsnamen. Der gespeicherte Wert wird am AR-Adapter-Ausgang `VALUEO` zur Verfügung gestellt. Er dient als variabler Operand für die nachfolgende arithmetische Verarbeitung.

Weitere Funktionsbausteine

- **DigitalInput_I1**
  - **Typ**: `logiBUS::io::DI::logiBUS_IXA`
  - Parameter: `QI` = `TRUE`, `Input` = `Input_I1`
  - Ereignisausgang/-eingang: AR-Adapter (`IN`)
  - Datenausgang/-eingang: –
- **AX_TON**
  - **Typ**: `adapter::events::unidirectional::timers::ATM_AX_TON`
  - Parameter: keine
  - Ereignisausgang/-eingang: `IN` (Adapter), `Q` (Adapter)
  - Datenausgang/-eingang: `PT` (Timer-Vorgabezeit) über AR-Adapter
- **AR_MULTIME**
  - **Typ**: `adapter::iec61131::arithmetic::AR_MULTIME`
  - Parameter: `IN1` = `T#100ms` (fester Multiplikator)
  - Datenausgang/-eingang: `IN2` (Multiplikand), `OUT` (Ergebnis) über AR-Adapter
- **DigitalOutput_Q1**
  - **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
  - Parameter: `QI` = `TRUE`, `Output` = `Output_Q1`
  - Ereignisausgang/-eingang: AR-Adapter (`OUT`)
  - Datenausgang/-eingang: –

Programmablauf und Verbindungen

1. **Digitaler Eingang** Der Baustein `DigitalInput_I1` stellt das physikalische Signal `Input_I1` (z. B. Taster oder Sensor) als AR-Adapter-Signal `IN` zur Verfügung.
2. **Timer-Start** Dieses Signal wird direkt mit dem `IN`-Adapter des Timers `AX_TON` verbunden.
   - Bei einer steigenden Flanke (EIN) startet der Timer.
3. **Variable Timer-Zeit** Der Sub-Baustein `Uebung_012e_sub_AR` liefert über seinen AR-Ausgang `VALUEO` den aus dem NVS geladenen numerischen Wert.
   - Dieser Wert wird mit dem AR-Adapter an den `IN2`-Eingang des arithmetischen Bausteins `AR_MULTIME` übergeben.

- Der Baustein `AR_MULTIME` multipliziert den festen Wert `T#100ms` (IN1) mit dem variablen Wert (IN2) und gibt das Ergebnis (Time) an seinem `OUT`-Adapter aus.
- Der Ausgang `OUT` wird mit dem `PT`-Adapter des Timers `AX_TON` verbunden. Dadurch wird die Timer-Ablaufzeit dynamisch aus dem gespeicherten Wert berechnet.

4. **Digitaler Ausgang** Der Timerausgang `Q` von `AX_TON` schaltet den `OUT`-Adapter des Ausgangsbausteins `DigitalOutput_Q1`. Somit wird der physikalische Ausgang `Output_Q1` aktiviert, solange der Timer läuft bzw. nach Ablauf der eingestellten Zeit.

Erläuterung der Verbindungen im Netzwerk:

- `DigitalInput_I1.IN` ⟶ `AX_TON.IN`
- `AX_TON.Q` ⟶ `DigitalOutput_Q1.OUT`
- `Uebung_012e_sub_AR.VALUEO` ⟶ `AR_MULTIME.IN2`
- `AR_MULTIME.OUT` ⟶ `AX_TON.PT`

Zusammenfassung

Lernziele:

- Einbindung eines AR-Adapter-basierten Sub-App-Bausteins zur persistenten Speicherung von Konfigurationswerten (NVS).
- Arithmetische Verknüpfung von Konstanten und gespeicherten Werten über AR-Adapter.
- Realisierung einer einstellbaren Timer-Funktion mit einem digitalen Eingang und Ausgang.
- Verständnis der Adapter-basierten Kommunikation zwischen Funktionsbausteinen unterschiedlicher Bibliotheken.

Schwierigkeitsgrad: Mittel **Benötigte Vorkenntnisse**: Grundlegende Kenntnisse der 4diac-IDE, Umgang mit logiBUS-Bausteinen, AR-Adapter und NVS-Konstanten. **Start der Übung**: Importieren Sie das SubApp-Template `Uebung_012e2_AR` in ein neues 4diac-Projekt, stellen Sie sicher, dass die benötigten Bibliotheken (`logiBUS`, `MyLib`, `Uebungen::const`) im Build-Pfad liegen, und verbinden Sie die physikalischen E/A-Punkte entsprechend der Hardware.

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

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

2.1.2.195 Uebung_012e_AR: Numeric Value Input PHYS und Speichern INI mit Subapp (AR Adapter)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert die Eingabe eines numerischen Werts (PHYS) und dessen Speicherung mittels eines SubApp, der einen AR-Adapter verwendet. Der eingegebene Wert wird unter einem vordefinierten Schlüssel in einem nichtflüchtigen Speicher (NVS) abgelegt. Die Übung zeigt, wie Konstanten aus einem zentralen Pool (`DefaultPool_Numeric`) sowie NVS-Schlüssel importiert und in einem SubApp-Parameter genutzt werden.

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: Uebung_012e_sub_AR

- **Typ**: `MyLib::sys::INI_IN_AND_STORE_AR`
- **Verwendete interne FBs**: Keine weiteren FBs im Netzwerk – der SubApp kapselt die gesamte Logik zur Initialisierung und Speicherung.
- **Parameter**:
  - `KEY` = `KEY_I1_STORE` (Konstante aus `Uebungen::const::NVS::NVS_Keys`)
  - `SECTION` = `SECTION_I1_STORE` (Konstante aus `Uebungen::const::NVS::NVS_Keys`)
  - `stObj` = `InputNumber_I1` (Konstante aus `Uebungen::const::UT::DefaultPool_Numeric`)
- **Funktionsweise**: Der SubApp `INI_IN_AND_STORE_AR` erwartet einen numerischen Eingangswert (z. B. über den AR-Adapter) und speichert diesen im nichtflüchtigen Speicher unter dem angegebenen Abschnitt und Schlüssel. Die Konstanten `KEY_I1_STORE` und `SECTION_I1_STORE` definieren den genauen Speicherort. Die Konstante `InputNumber_I1` stellt den Initialwert oder die Struktur des zu speichernden Objekts bereit.

Programmablauf und Verbindungen

- **Lernziele**:
  - Anwendung von AR-Adaptern in SubApps
  - Umgang mit NVS-Konstanten und Konstanten-Pools
  - Parametrisierung von SubApps zur Konfiguration von Speicherorten
- **Schwierigkeitsgrad**: Mittel
- **Vorkenntnisse**: Grundlagen der 4diac-IDE, Erstellung von SubApps, Verständnis von AR-Schnittstellen und nichtflüchtigen Speichern.

Die Übung besteht aus einem einzigen SubApp, der in ein übergeordnetes System eingebunden wird. Der SubApp wird mit den drei Parametern konfiguriert. Die Verbindung des AR-Adapters und die Bereitstellung des numerischen Werts erfolgen außerhalb dieser SubApp-Definition (z. B. in der übergeordneten Applikation). Die gespeicherten Daten können anschließend über die gleichen Schlüssel wieder ausgelesen werden.

Zusammenfassung

Die Übung `Uebung_012e_AR` zeigt eine kompakte Lösung zur persistierenden Speicherung eines numerischen Wertes unter Verwendung eines SubApp mit AR-Adapter. Durch die Auslagerung der Speicherlogik in den wiederverwendbaren SubApp `INI_IN_AND_STORE_AR` und die zentrale Definition von Schlüsseln und Objekten wird eine klare Trennung von Konfiguration und Funktionslogik erreicht.

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

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

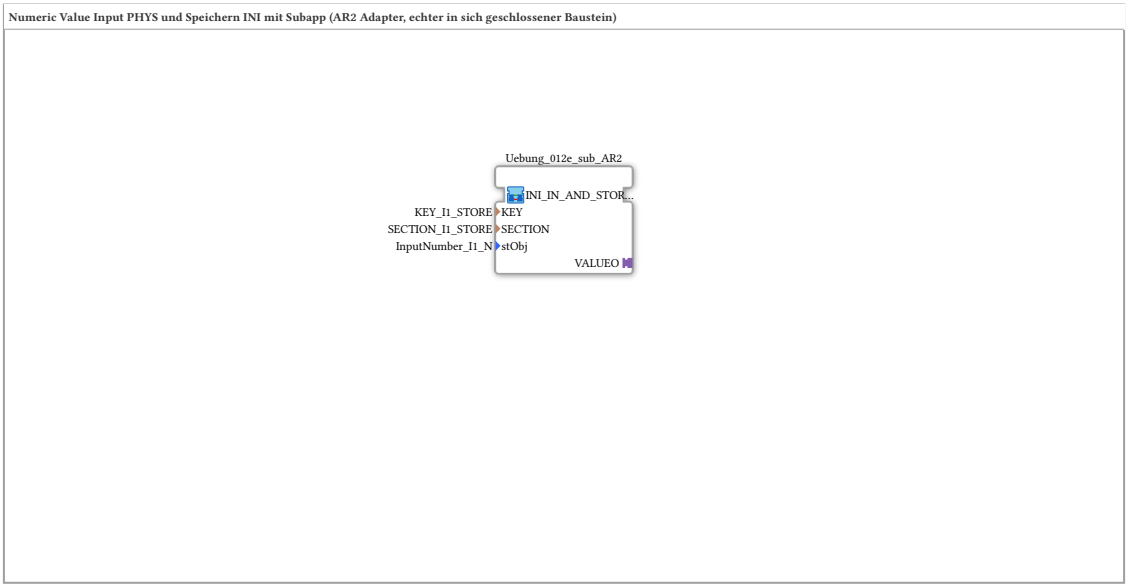
2.1.2.196 Uebung_012e_AR2: Numeric Value Input PHYS und Speichern INI via AR2 mit Subapp

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_012e_AR2`. Teil einer Serie von Übungen (`Uebung_012d_AR2` bis `Uebung_012o_AR2`), die verschiedene Kombinationen aus Eingabequelle (VT-Feld, OPC-UA oder beides) und Speicherziel (NVS-Flash oder INI-Datei) über den bidirektionalen `AR2`-Adapter demonstrieren.

Ziel der Übung

Einen numerischen Wert aus einem VT-Zahlenfeld (`NumericValue_TO_AR2`) entgegennehmen und dauerhaft in INI-Datei speichern - unter Verwendung des bidirektionalen AR2-Adapters (als fertig gekapselte Subapp).

Beschreibung und Komponenten



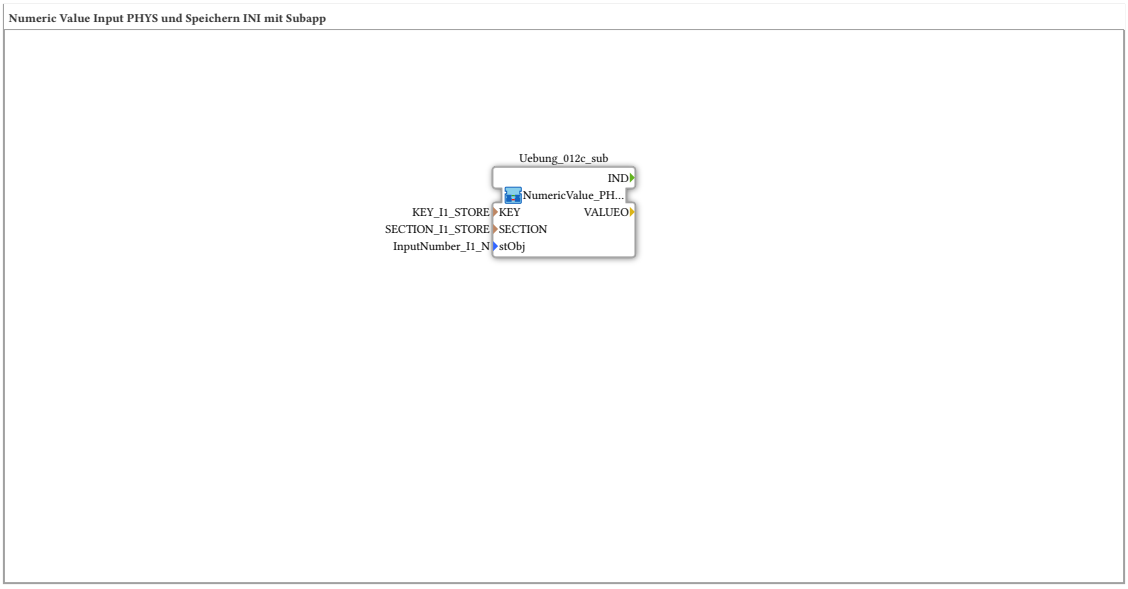
- **`INI_IN_AND_STORE_AR2 (Subapp)`**: kapselt VT-/OPC-UA-Eingang, den AR2-Adapter und die Speicherung in INI-Datei als ein einziger, in sich geschlossener Baustein - die interne Verdrahtung ist nicht als separates FB-Netzwerk sichtbar, sondern vollständig im Baustein selbst gekapselt.

Funktionsweise

Der Zahlenwert kommt aus einem VT-Zahlenfeld (`NumericValue_TO_AR2`) und wird von `INI_IN_AND_STORE_AR2 (Subapp)` intern verarbeitet und in INI-Datei gespeichert, ohne dass die einzelnen Verarbeitungsschritte als separate FB-Instanzen in dieser Subapplikation sichtbar sind.

2.1.2.197 Uebung_012e_AX: Numeric Value Input PHYS und Speichern INI mit Subapp

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_012e_AX (Numeric Value Input PHYS und Speichern INI mit Subapp).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Numeric Value Input PHYS und Speichern INI mit Subapp**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_012e_AX.SUB, welche die folgende Bausteinstruktur verwendet:

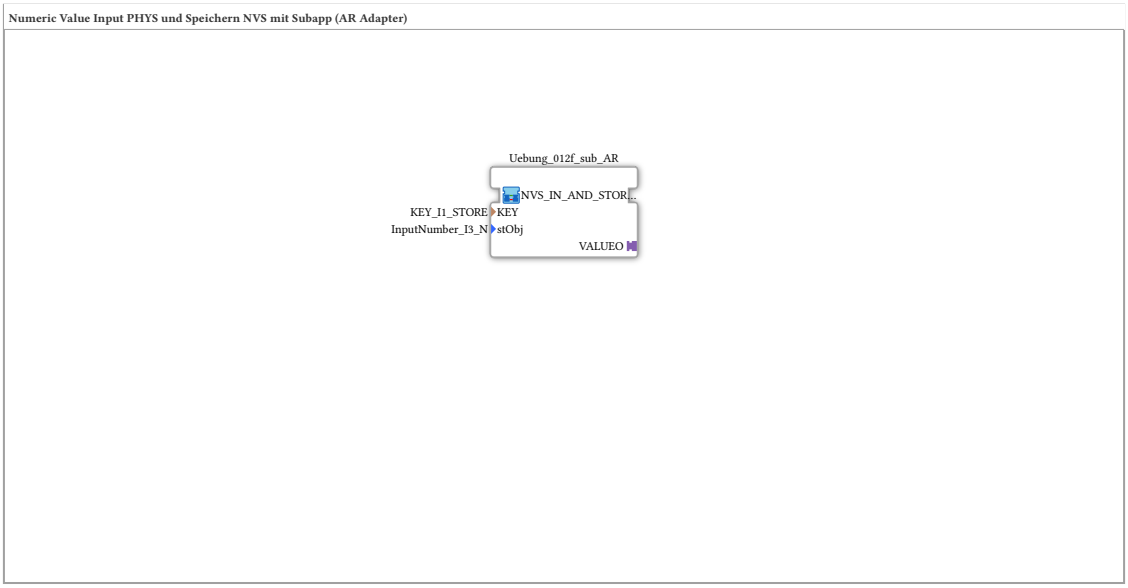
Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Signalweiterleitung**: Die Bausteine verarbeiten Daten- und Steuersignale gemäß ihrer Konfiguration.

Zusammenfassung

Die Übung Uebung_012e_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.198 Uebung_012f_AR: Numeric Value Input PHYS und Speichern NVS mit Subapp (AR Adapter)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung befasst sich mit dem Einlesen eines numerischen Werts (physikalische Größe) und dessen Speicherung im nichtflüchtigen Speicher (NVS) unter Verwendung eines AR-Adapters. Die SubApp dient als Baustein für Anwendungen, die einen Wert einmalig einlesen und dauerhaft ablegen müssen.

Verwendete Funktionsbausteine (FBs)

Sub-Baustein: `Uebung_012f_sub_AR`

- **Typ**: `MyLib::sys::NVS_IN_AND_STORE_AR`
- **Parameter**:
  - `KEY` = `KEY_I1_STORE`
  - `stObj` = `InputNumber_I3`
- **Funktionsweise**: Der Baustein nimmt über den eingebauten AR-Adapter einen numerischen Wert entgegen. Der Datentyp und die Struktur des Werts sind durch die vorgegebene Konstante `InputNumber_I3` aus der Bibliothek `DefaultPool_Numeric` definiert. Der gelesene Wert wird unter dem Speicherschlüssel `KEY_I1_STORE` aus der Konstantenbibliothek `NVS_Keys` im nichtflüchtigen Speicher abgelegt.

Programmablauf und Verbindungen

Die SubApp `Uebung_012f_AR` selbst besitzt keine eigenen Ein- oder Ausgangsschnittstellen. Sie instanziiert den fertigen Subbaustein `Uebung_012f_sub_AR` und versorgt ihn mit den beiden Parametern `KEY` und `stObj`. Diese Parameter werden aus den importierten Bibliotheken mit den entsprechenden Konstanten belegt. Der Baustein führt dann die gesamte Logik zum Empfang des Werts über den AR-Adapter und zum Speichern im NVS aus.

Lernziele dieser Übung:

- Verstehen, wie ein AR-Adapter für die Werteingabe verwendet wird.
- Speichern von Werten im nichtflüchtigen Speicher mit vordefinierten Schlüsseln.
- Verwaltung von Konstanten aus externen Bibliotheken.

Schwierigkeitsgrad: Mittel **Vorkenntnisse:** Grundlegendes Verständnis von 4diac-FB-Netzwerken, SubApps und der Verwendung von Adaptern.

Zusammenfassung

Die Übung `Uebung_012f_AR` demonstriert das persistente Ablegen eines numerischen Werts mithilfe eines AR-Adapters. Der vorgefertigte Baustein `NVS_IN_AND_STORE_AR` wird parametriert und in einer SubApp gekapselt, wodurch eine klare und wiederverwendbare Lösung für das Einlesen und Speichern von Daten entsteht.

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

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

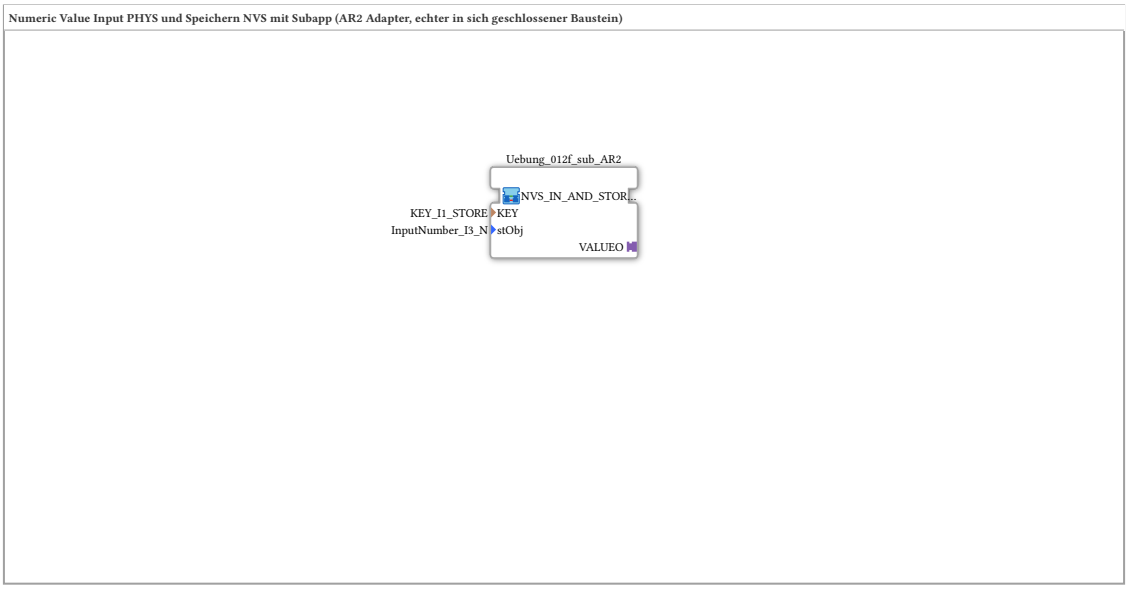
2.1.2.199 Uebung_012f_AR2: Numeric Value Input PHYS und Speichern NVS via AR2 mit Subapp

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_012f_AR2`. Teil einer Serie von Übungen (`Uebung_012d_AR2` bis `Uebung_012o_AR2`), die verschiedene Kombinationen aus Eingabequelle (VT-Feld, OPC-UA oder beides) und Speicherziel (NVS-Flash oder INI-Datei) über den bidirektionalen `AR2`-Adapter demonstrieren.

Ziel der Übung

Einen numerischen Wert aus einem VT-Zahlenfeld (`NumericValue_TO_AR2`) entgegennehmen und dauerhaft in NVS (ESP32-Flash) speichern - unter Verwendung des bidirektionalen AR2-Adapters (als fertig gekapselte Subapp).

Beschreibung und Komponenten



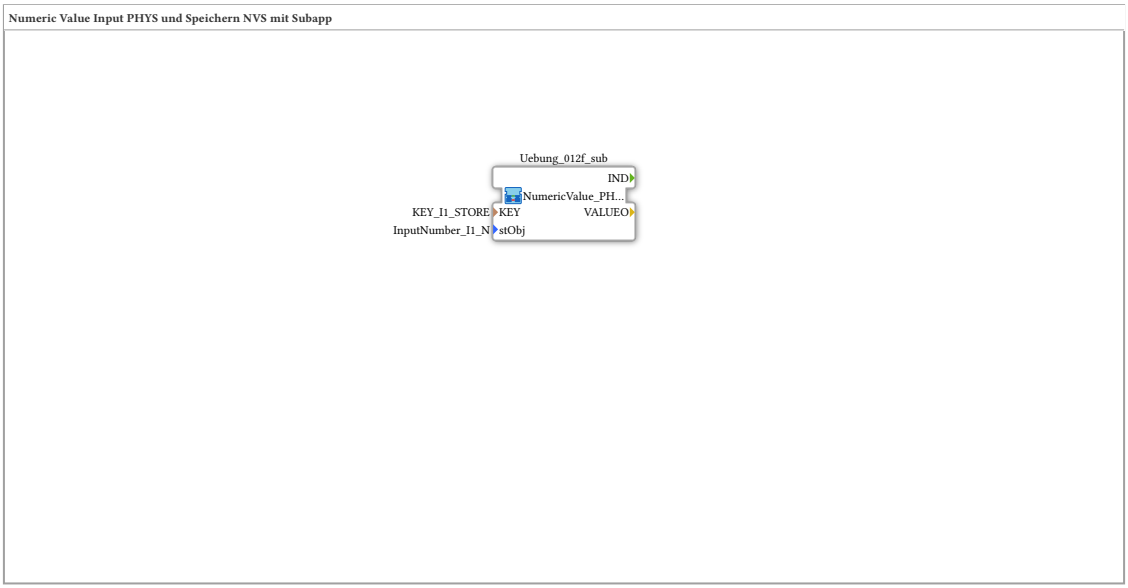
- **`NVS_IN_AND_STORE_AR2 (Subapp)`**: kapselt VT-/OPC-UA-Eingang, den AR2-Adapter und die Speicherung in NVS (ESP32-Flash) als ein einziger, in sich geschlossener Baustein - die interne Verdrahtung ist nicht als separates FB-Netzwerk sichtbar, sondern vollständig im Baustein selbst gekapselt.

Funktionsweise

Der Zahlenwert kommt aus einem VT-Zahlenfeld (`NumericValue_TO_AR2`) und wird von `NVS_IN_AND_STORE_AR2 (Subapp)` intern verarbeitet und in NVS (ESP32-Flash) gespeichert, ohne dass die einzelnen Verarbeitungsschritte als separate FB-Instanzen in dieser Subapplikation sichtbar sind.

2.1.2.200 Uebung_012f_AX: Numeric Value Input PHYS und Speichern NVS mit Subapp

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_012f_AX (Numeric Value Input PHYS und Speichern NVS mit Subapp).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Numeric Value Input PHYS und Speichern NVS mit Subapp**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_012f_AX.SUB, welche die folgende Bausteinstruktur verwendet:

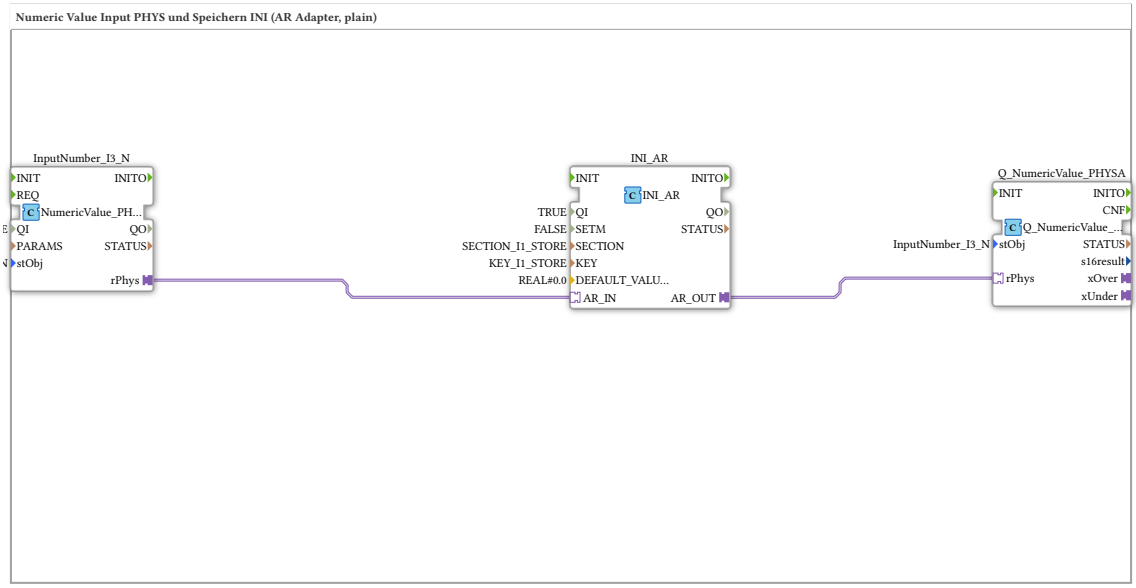
Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Signalweiterleitung**: Die Bausteine verarbeiten Daten- und Steuersignale gemäß ihrer Konfiguration.

Zusammenfassung

Die Übung Uebung_012f_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.201 Uebung_012g_AR: Numeric Value Input PHYS und Speichern INI (AR Adapter, plain)



• – ○ • – ○ • – ○ •

Einleitung

In dieser Übung wird ein numerischer Wert (REAL) über einen physikalischen Eingang eingelesen und mittels des INI_AR-Adapters dauerhaft gespeichert. Der gespeicherte Wert kann anschließend über einen weiteren Ausgabebaustein visualisiert werden. Die Übung demonstriert den Einsatz des AR-Adapter-Interfaces (Adapter-Ressource) zur Kommunikation zwischen einem physikalischen Eingabebaustein und einem Speicherbaustein sowie einer numerischen Ausgabe.

Verwendete Funktionsbausteine (FBs)

Es werden folgende Funktionsbausteine verwendet. Die SubApp enthält keine weiteren Sub-Bausteine.

FB: `InputNumber_I3`

- **Typ**: `isobus::UT::io::NumericValue::NumericValue_PHYSA`
- **Parameter**:
  - `QI` = `TRUE` (Initialwert für die Qualität des Eingangs)
  - `stObj` = `InputNumber_I3` (Referenz auf das physikalische Eingangsobjekt)
- **Adapter**:
  - Ausgang-Adapter (`rPhys`) für die Übergabe des numerischen Werts
- **Funktionsweise**: Der Baustein liest einen numerischen Wert (REAL) von der konfigurierten physikalischen Eingangsquelle ein. Der Wert wird über den `rPhys`-Adapterausgang an nachfolgende Bausteine weitergegeben.

FB: `INI_AR`

- **Typ**: `eclipse4diac::storage::INI_AR`
- **Parameter**:
  - `QI` = `TRUE` (Aktivierung des Speichervorgangs)
  - `KEY` = `KEY_I1_STORE` (Speicherschlüssel, importiert aus `Uebungen::const::NVS::NVS_Keys`)
  - `DEFAULT_VALUE` = `REAL#0.0` (Standardwert, falls noch kein Wert gespeichert wurde)
- **Adapter**:
  - Eingang-Adapter (`AR_IN`) für den zu speichernden Wert
  - Ausgang-Adapter (`AR_OUT`) für den ausgelesenen gespeicherten Wert
- **Funktionsweise**: Der Baustein speichert einen vom `AR_IN`-Adapter empfangenen Wert unter dem angegebenen Schlüssel im nichtflüchtigen Speicher (NVS). Beim Start oder nach dem Speichern wird

der gespeicherte Wert am AR_OUT-Adapter bereitgestellt. Der DEFAULT_VALUE wird verwendet, wenn noch kein Wert hinterlegt ist.

FB: Q_NumericValue

- **Typ**: isobus::UT::Q::Q_NumericValue_PHYSA
- **Parameter**:
  - stObj = InputNumber_I3 (Referenz auf dasselbe physikalische Objekt wie beim Eingang)
- **Adapter**:
  - Eingang-Adapter (rPhys) für den anzuzeigenden Wert
- **Funktionsweise**: Der Baustein gibt den über den rPhys-Adapter empfangenen numerischen Wert an die konfigurierte Ausgabe (z. B. eine Anzeige oder einen virtuellen Ausgang) weiter. Er dient zur Visualisierung oder Weiterverarbeitung des aktuellen oder gespeicherten Werts.

Programmablauf und Verbindungen

Die Verbindung der Bausteine erfolgt ausschließlich über **Adapter**. Das Netzwerk besteht aus drei Bausteinen, die wie folgt miteinander verbunden sind:

1. **InputNumber_I3.rPhys → INI_AR.AR_IN** Der vom Eingabebaustein gelesene physikalische Wert wird direkt an den INI_AR-Speicherbaustein übergeben.
2. **INI_AR.AR_OUT → Q_NumericValue.rPhys** Der aus dem Speicher zurückgelesene Wert (entweder der neu gespeicherte oder der zuletzt gespeicherte) wird an den Ausgabebaustein weitergeleitet.

Damit wird eine einfache Pipeline realisiert: **Einlesen → Speichern → Ausgeben**

Die Funktionsweise ist ereignisgesteuert (im Hintergrund durch die Adapter). Die Parameter QI beider Bausteine sind dauerhaft auf TRUE gesetzt, sodass der Datenfluss kontinuierlich erfolgt.

Lernziele:

- Verständnis des AR-Adapter-Interfaces für die Kommunikation zwischen Bausteinen
- Einrichtung eines persistenten Speichers mit dem INI_AR-Baustein
- Zusammenspiel von physikalischem Eingang, Speicher und Ausgabe

Schwierigkeitsgrad: Mittel **Vorkenntnisse:** Grundlagen der 4diac-IDE, Umgang mit Funktionsbausteinen und Adaptern

Zusammenfassung

Die Übung Uebung_012g_AR zeigt eine kompakte Realisierung eines numerischen Wertespeichers unter Verwendung des AR-Adapter-Konzepts. Der Wert wird von einem physikalischen Eingang gelesen, über den INI_AR-Baustein persistiert und anschließend über einen Ausgabebaustein sichtbar gemacht. Die Lösung besteht aus drei spezialisierten Funktionsbausteinen, die über Adapter verbunden sind und so eine saubere Trennung von Ein-, Speicher- und Ausgabelogik ermöglichen.

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

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

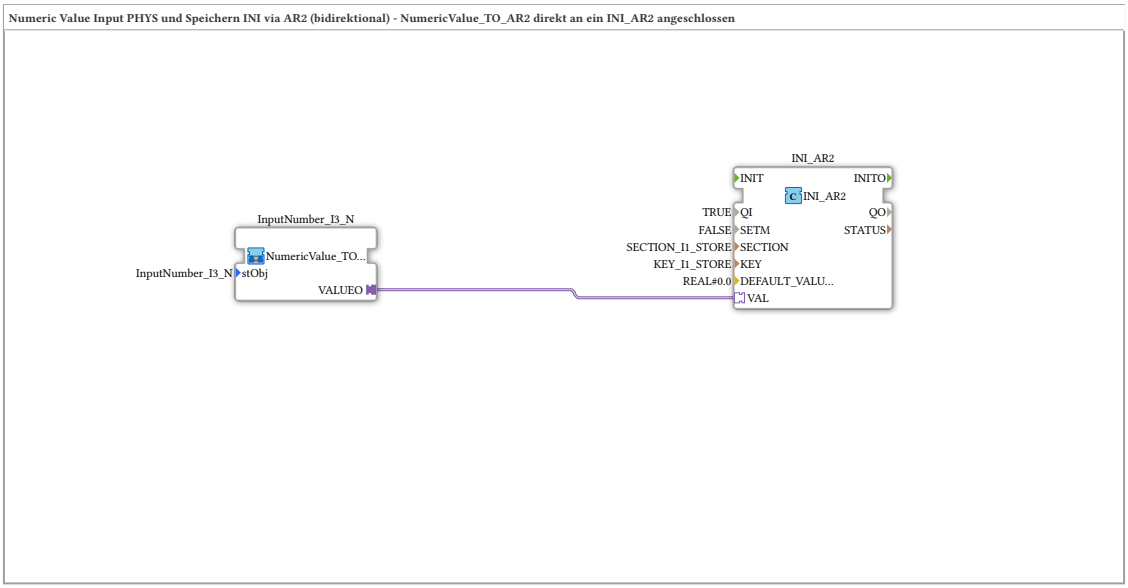
2.1.2.202 Uebung_012g_AR2: Numeric Value Input PHYS und Speichern INI via AR2 (bidirektional, direkt)

Dieser Artikel beschreibt die logiBUS®-Übung Uebung_012g_AR2. Teil einer Serie von Übungen (Uebung_012d_AR2 bis Uebung_012o_AR2), die verschiedene Kombinationen aus Eingabequelle (VT-Feld, OPC-UA oder beides) und Speicherziel (NVS-Flash oder INI-Datei) über den bidirektionalen AR2-Adapter demonstrieren.

Ziel der Übung

Einen numerischen Wert aus einem VT-Zahlenfeld (NumericValue_TO_AR2) entgegennehmen und dauerhaft in INI-Datei speichern - unter Verwendung des bidirektionalen AR2-Adapters (direkt verdrahtet, ohne Subapp-Kapselung).

Beschreibung und Komponenten



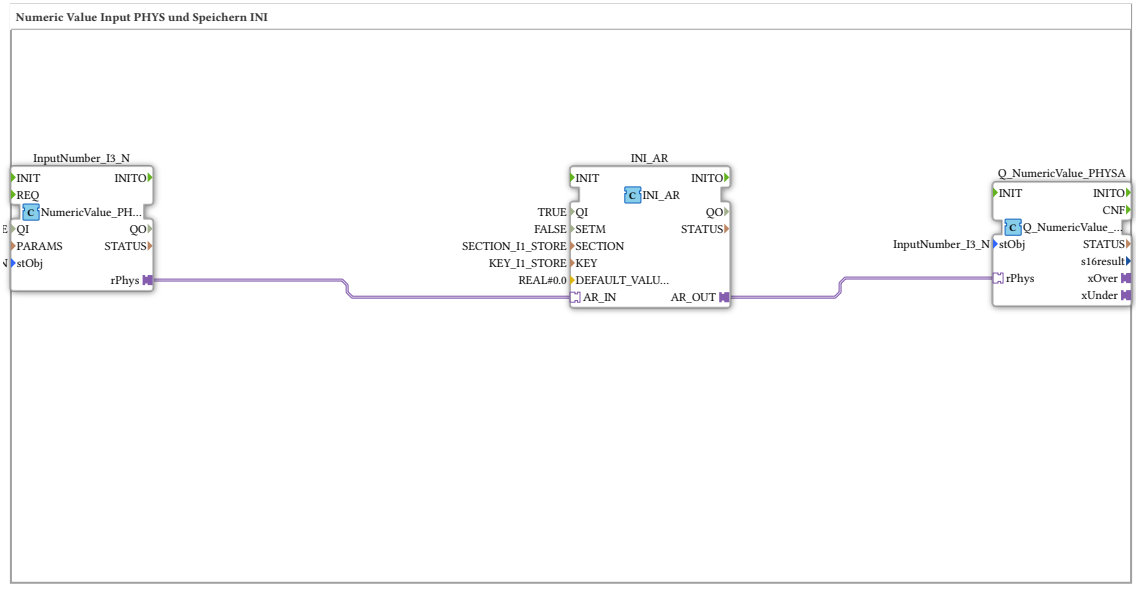
- **`INI_AR2`**: `eclipse4diac::storage::INI_AR2`, bidirektionaler AR2-Adapter-Baustein, der den Wert direkt persistiert.

Funktionsweise

Der Zahlenwert wird aus einem VT-Zahlenfeld (`NumericValue_TO_AR2`) direkt über den bidirektionalen `AR2`-Adapter an `INI_AR2` weitergegeben, der ihn in INI-Datei speichert - ohne Zwischenschritt über eine separate Subapp.

2.1.2.203 Uebung_012g_AX: Numeric Value Input PHYS und Speichern INI

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_012g_AX (Numeric Value Input PHYS und Speichern INI).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Numeric Value Input PHYS und Speichern INI**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_012g_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **InputNumber_I3_N**: Instanz des Typs isobus::UT::io::NumericValue::NumericValue_PHYSA.
  - Parameter QI = TRUE
  - Parameter stObj = InputNumber_I3_N
- **INI_AR**: Instanz des Typs eclipse4diac::storage::INI_AR.
  - Parameter QI = TRUE
  - Parameter SETM = FALSE
  - Parameter SECTION = SECTION_I1_STORE
  - Parameter KEY = KEY_I1_STORE
  - Parameter DEFAULT_VALUE = REAL#0.0
- **Q_NumericValue_PHYSA**: Instanz des Typs isobus::UT::Q::Q_NumericValue_PHYSA.
  - Parameter stObj = InputNumber_I3_N

Verbindungen und Schnittstellen

Adapterverbindungen:

- INI_AR.AR_OUT -> Q_NumericValue_PHYSA.rPhys
- InputNumber_I3_N.rPhys -> INI_AR.AR_IN

Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_012g_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

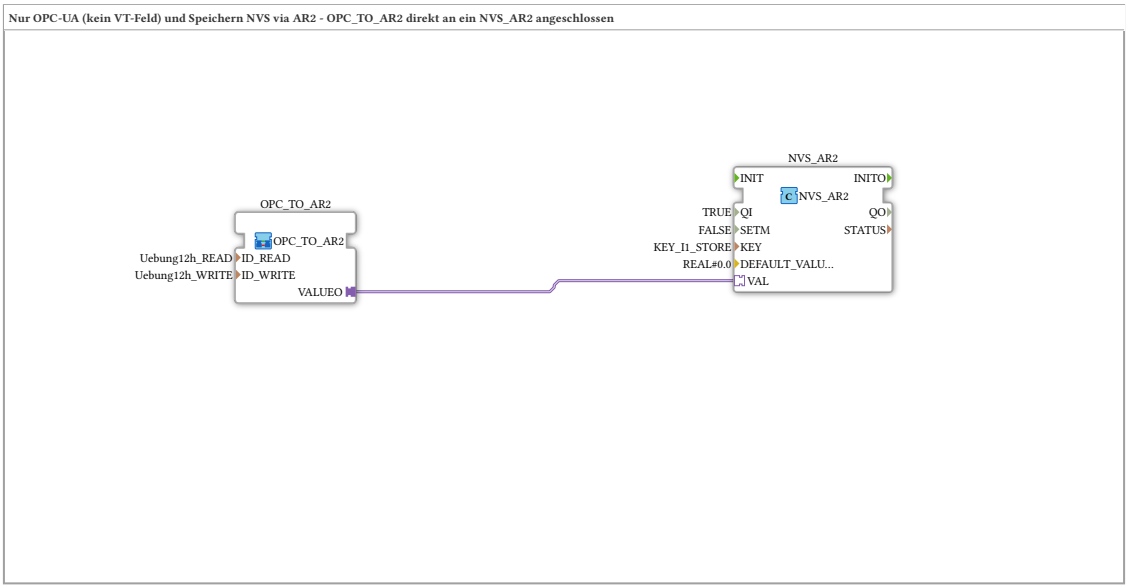
2.1.2.204 Uebung_012h_AR2: Numeric Value Input (kein VT-Feld) und Speichern NVS via AR2 (bidirektional, direkt)

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_012h_AR2`. Teil einer Serie von Übungen (`Uebung_012d_AR2` bis `Uebung_012o_AR2`), die verschiedene Kombinationen aus Eingabequelle (VT-Feld, OPC-UA oder beides) und Speicherziel (NVS-Flash oder INI-Datei) über den bidirektionalen `AR2`-Adapter demonstrieren.

Ziel der Übung

Einen numerischen Wert ausschließlich über OPC-UA (`OPC_TO_AR2`), es gibt kein VT-Zahlenfeld auf der Maske entgegennehmen und dauerhaft in NVS (ESP32-Flash) speichern - unter Verwendung des bidirektionalen AR2-Adapters (direkt verdrahtet, ohne Subapp-Kapselung).

Beschreibung und Komponenten



- **`NVS_AR2`**: `logiBUS::storage::esp32_nvs::NVS_AR2`, bidirektionaler AR2-Adapter-Baustein, der den Wert direkt persistiert.

Funktionsweise

Der Zahlenwert wird ausschließlich über OPC-UA (`OPC_TO_AR2`), es gibt kein VT-Zahlenfeld auf der Maske direkt über den bidirektionalen `AR2`-Adapter an `NVS_AR2` weitergegeben, der ihn in NVS (ESP32-Flash) speichert - ohne Zwischenschritt über eine separate Subapp.

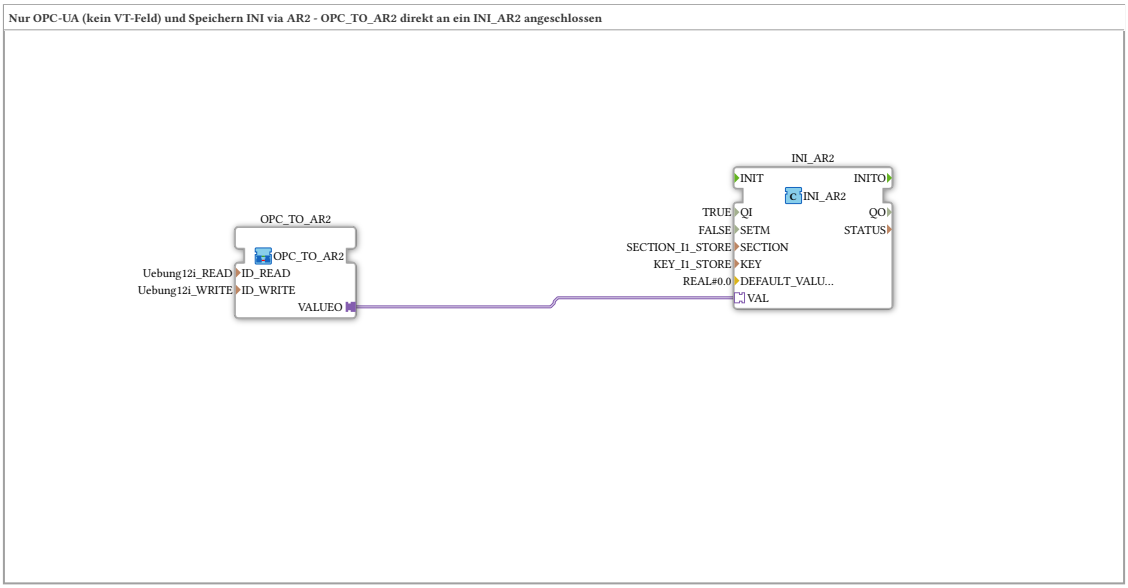
2.1.2.205 Uebung_012i_AR2: Numeric Value Input (kein VT-Feld) und Speichern INI via AR2 (bidirektional, direkt)

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_012i_AR2`. Teil einer Serie von Übungen (`Uebung_012d_AR2` bis `Uebung_012o_AR2`), die verschiedene Kombinationen aus Eingabequelle (VT-Feld, OPC-UA oder beides) und Speicherziel (NVS-Flash oder INI-Datei) über den bidirektionalen `AR2`-Adapter demonstrieren.

Ziel der Übung

Einen numerischen Wert ausschließlich über OPC-UA (`OPC_TO_AR2`), es gibt kein VT-Zahlenfeld auf der Maske entgegennehmen und dauerhaft in INI-Datei speichern - unter Verwendung des bidirektionalen AR2-Adapters (direkt verdrahtet, ohne Subapp-Kapselung).

Beschreibung und Komponenten

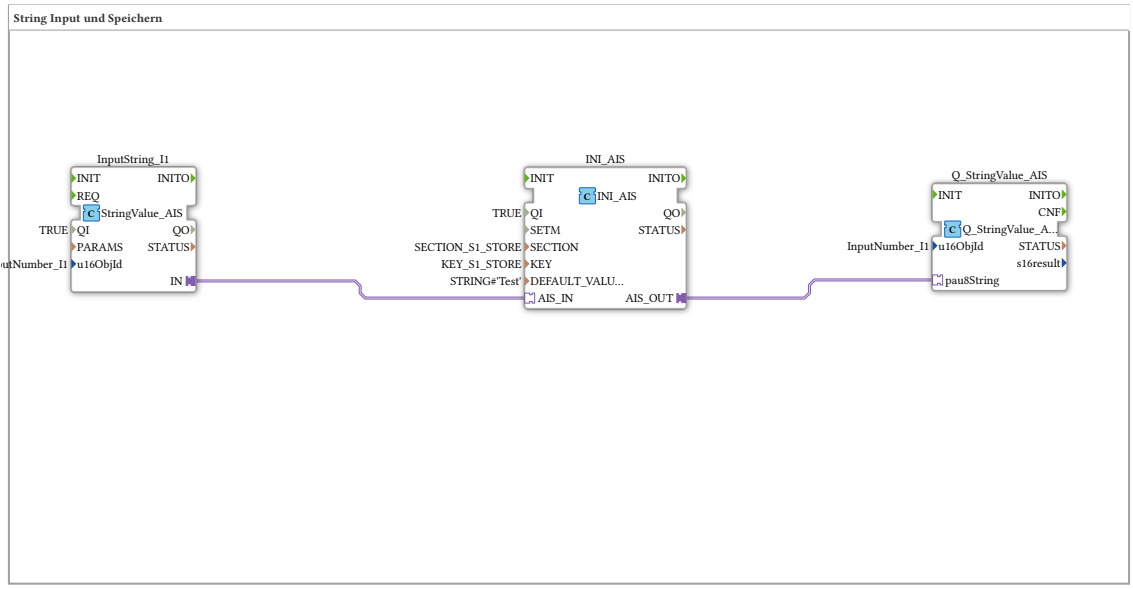


- **`INI_AR2`**: `eclipse4diac::storage::INI_AR2`, bidirektionaler AR2-Adapter-Baustein, der den Wert direkt persistiert.

Funktionsweise

Der Zahlenwert wird ausschließlich über OPC-UA (`OPC_TO_AR2`), es gibt kein VT-Zahlenfeld auf der Maske direkt über den bidirektionalen `AR2`-Adapter an `INI_AR2` weitergegeben, der ihn in INI-Datei speichert - ohne Zwischenschritt über eine separate Subapp.

2.1.2.206 Uebung_012j_AIS: String Input und Speichern



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert das Einlesen eines String-Wertes über einen virtuellen Eingang, das Speichern in einem nichtflüchtigen Speicher (NVS) und die anschließende Ausgabe des gespeicherten Wertes. Sie zeigt den Umgang mit den Adapter-Schnittstellen der 4diac-IDE und die Nutzung vordefinierter Konstanten für Speicherbereiche.

Ziel ist es, einen Anfangswert (Default: „Test“) im NVS zu speichern und diesen über einen Funktionsbaustein auszugeben, wobei der Wert bei jedem Neustart aus dem Speicher geladen wird.

Verwendete Funktionsbausteine (FBs)

Die Übung besteht aus drei Funktionsbausteinen, die über Adapter verbunden sind:

FB: InputString_I1

- **Typ**: `isobus::UT::io::StringValue::StringValue_AIS`
- **Parameter**:
  - `QI` = `TRUE` (Baustein aktiv)
  - `u16ObjId` = `InputNumber\_I1` (Konstante zur Identifikation des Eingangsobjekts)
- **Ereignisausgang/-eingang**: Standard INIT/REQT (nicht näher konfiguriert)
- **Datenausgang/-eingang**: Ausgang `IN` (Adapterausgang, liefert den eingegebenen String)
- **Funktionsweise**: Dieser Baustein liest einen String-Wert von einer virtuellen Eingabestelle (z. B. HMI oder Simulation) und stellt ihn über den Adapter `IN` bereit. Der Wert wird durch die Objekt-ID `InputNumber_I1` identifiziert.

FB: INI_AIS

- **Typ**: `eclipse4diac::storage::INI_AIS`
- **Parameter**:
  - `QI` = `TRUE` (Baustein aktiv)
  - `SECTION` = `SECTION_S1_STORE` (Konstante für den Speicherabschnitt)
  - `KEY` = `KEY_S1_STORE` (Konstante für den Speicherschlüssel)
  - `DEFAULT_VALUE` = `STRING#'Test'` (Standardwert, falls noch kein Wert gespeichert ist)
- **Ereignisausgang/-eingang**: Standard INIT/REQT
- **Datenausgang/-eingang**:
  - `AIS_IN` (Adaptereingang, erwartet einen String)
  - `AIS_OUT` (Adapterausgang, gibt den gespeicherten oder geladenen String aus)
- **Funktionsweise**: Der Baustein fungiert als Speicherzugriff für nichtflüchtigen Speicher (NVS). Er speichert einen über `AIS_IN` erhaltenen String unter der angegebenen Sektion und dem Schlüssel. Wird kein neuer Wert zugeführt, liefert er den zuletzt gespeicherten Wert oder den `DEFAULT_VALUE` über `AIS_OUT`.

FB: Q_StringValue_AIS

- **Typ**: `isobus::UT::Q::Q_StringValue_AIS`
- **Parameter**:
  - `u16ObjId` = `InputNumber_I1` (gleiche Objekt-ID wie Eingang)
- **Ereignisausgang/-eingang**: Standard INIT/REQT
- **Datenausgang/-eingang**:
  - `pau8String` (Adaptereingang, erwartet den anzuzeigenden String)
- **Funktionsweise**: Dieser Baustein gibt den über `pau8String` erhaltenen String an eine Ausgabestelle (z. B. Anzeige, übergeordnete Applikation) weiter. Die Ausgabe erfolgt anhand der Objekt-ID `InputNumber_I1`.

Programmablauf und Verbindungen

Der Datenfluss innerhalb der SubApp erfolgt über Adapterverbindungen:

1. **Eingabe**: `InputString_I1` stellt den aktuellen String (vom HMI oder aus einer Simulation) an seinem Adapterausgang `IN` bereit.
2. **Speichern**: Der Adapterausgang `IN` wird mit dem Adaptereingang `AIS_IN` von `INI_AIS` verbunden. `INI_AIS` speichert diesen Wert im nichtflüchtigen Speicher (Sektion und Schlüssel gemäß den Konstanten).
3. **Ausgabe**: Der gespeicherte (oder geladene) Wert wird von `INI_AIS` über den Adapterausgang `AIS_OUT` bereitgestellt. Dieser ist mit dem Adaptereingang `pau8String` von `Q_StringValue_AIS` verbunden, der den Wert an die Ausgabestelle weitergibt.

Die SubApp ist so konzipiert, dass bei jedem Start automatisch der zuletzt gespeicherte Wert aus dem NVS geladen und ausgegeben wird (DEFAULT_VALUE dient als Initialwert). Eine externe Applikation kann den Eingangswert überschreiben, woraufhin der neue Wert gespeichert und sofort ausgegeben wird.

Hinweise:

- Die Konstanten `InputNumber_I1`, `SECTION_S1_STORE` und `KEY_S1_STORE` sind in übergeordneten Bibliotheken definiert und müssen vor der Verwendung importiert werden.
- Die Übung demonstriert ein typisches Muster für persistente Datenspeicherung in der Automatisierungstechnik.

Zusammenfassung

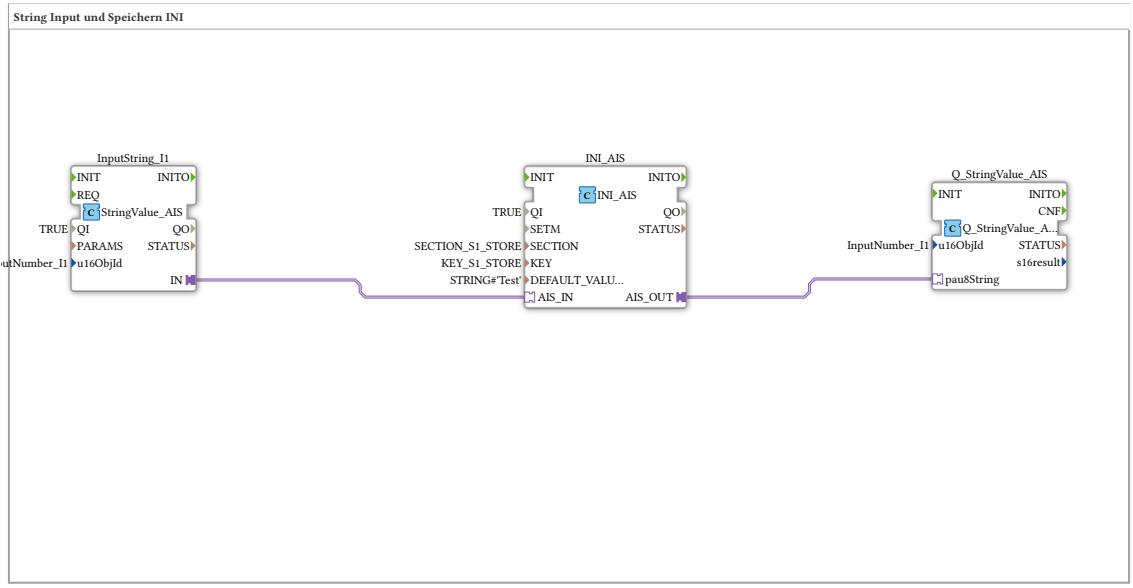
Die Übung **Uebung_012j_AIS** zeigt die grundlegende Handhabung von String-Daten in Kombination mit nichtflüchtigem Speicher (NVS) in der 4diac-IDE. Durch die Verwendung von Adapterverbindungen wird ein datenbankähnlicher Zugriff auf gespeicherte Werte realisiert. Die drei Funktionsbausteine – Eingabe, Speicher und Ausgabe – bilden eine wiederverwendbare Komponente, die in verschiedenen Applikationen zur persistenten Speicherung von Benutzereingaben eingesetzt werden kann. Die vorgegebenen Konstanten sorgen für eine klare Trennung der Speicherbereiche und erleichtern die Wartbarkeit.

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

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

2.1.2.207 Uebung_012j_AX: String Input und Speichern INI

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_012j_AX (String Input und Speichern INI).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **String Input und Speichern INI**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_012j_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **InputString_I1**: Instanz des Typs isobus::UT::io::StringValue::StringValue_AIS.
  - Parameter QI = TRUE
  - Parameter u16ObjId = InputNumber_I1
- **INI_AIS**: Instanz des Typs eclipse4diac::storage::INI_AIS.
  - Parameter QI = TRUE
  - Parameter SECTION = SECTION_S1_STORE
  - Parameter KEY = KEY_S1_STORE
  - Parameter DEFAULT_VALUE = STRING#‚Test‘
- **Q_StringValue_AIS**: Instanz des Typs isobus::UT::Q::Q_StringValue_AIS.
  - Parameter u16ObjId = InputNumber_I1

Verbindungen und Schnittstellen

Adapterverbindungen:

- InputString_I1.IN -> INI_AIS.AIS_IN
- INI_AIS.AIS_OUT -> Q_StringValue_AIS.pau8String

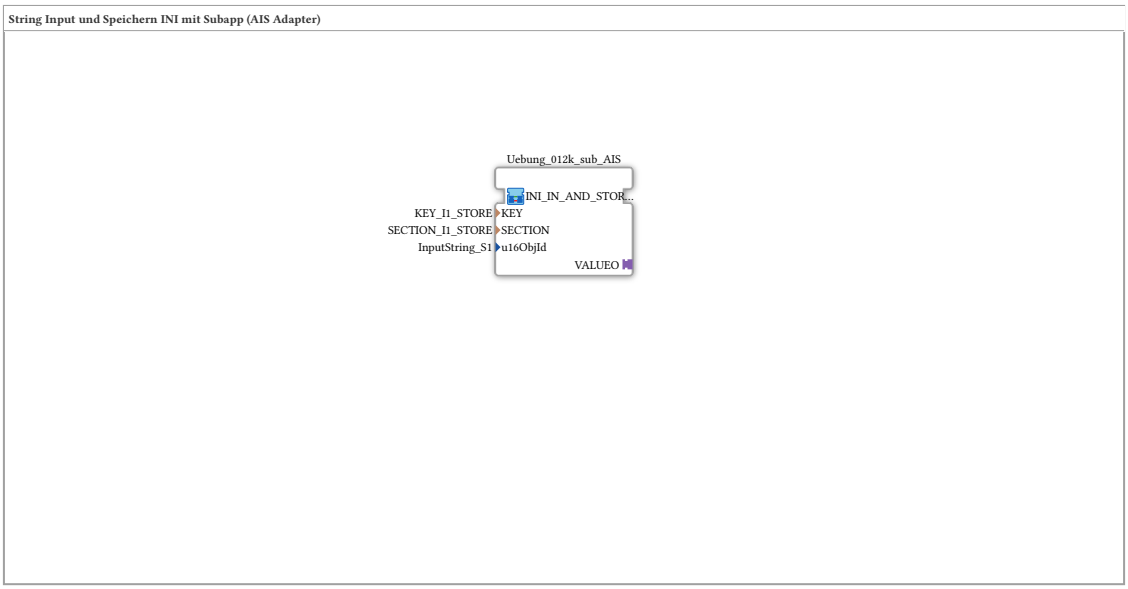
Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_012j_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.208 Uebung_012k_AIS: String Input und Speichern INI mit Subapp (AIS Adapter)



• – ◦ • – ◦ • – ◦ •

Einleitung

Die Übung *Uebung_012k_AIS* demonstriert, wie ein eingegebener String (z. B. Benutzereingabe oder Sensorwert) in einem nichtflüchtigen Speicher (NVS) abgelegt wird. Der Zugriff erfolgt über einen AIS-Adapter (Application Interface Subapp), der die Kommunikation mit der Speicherlogik vereinfacht. Ziel ist es, den Umgang mit persistenten Daten und Adapter-basierten Subapplikationen in der 4diac-IDE zu erlernen.

Verwendete Funktionsbausteine (FBs)

Die Übung besteht aus einer einzigen Subapplikation, die alle notwendigen Funktionen kapselt.

Sub-Baustein: Uebung_012k_sub_AIS

- **Typ**: `MyLib::sys::INI_IN_AND_STORE_AIS`
- **Verwendete interne FBs**: nicht einsehbar (intern durch den Typ definiert)
- **Parameter**:
  - `KEY` = `KEY_I1_STORE` (Konstante aus `Uebungen::const::NVS::NVS_Keys`)
  - `SECTION` = `SECTION_I1_STORE` (Konstante aus `Uebungen::const::NVS::NVS_Keys`)
  - `u16ObjId` = `InputString_S1` (Konstante aus `Uebungen::const::UT::DefaultPool`)

- **Funktionsweise**: Dieser Sub-Baustein empfängt über einen AIS-Adapter einen String-Eingangswert. Beim Eintreffen eines entsprechenden Ereignisses (vermutlich vom Adapter) wird der Wert unter der angegebenen Section und dem Key im NVS gespeichert. Die Objekt-ID (`u16ObjId`) identifiziert dabei den konkreten Eingangskanal. Nach erfolgreichem Speichern wird ein Bestätigungssignal (z. B. `INIT` oder `CNF`) zurückgegeben.

Programmablauf und Verbindungen

1. **Initialisierung**: Die Subapp `Uebung_012k_sub_AIS` wird mit den vordefinierten Konstanten parametriert.
2. **Datenfluss**: Ein String wird über den AIS-Adapter an den Sub-Baustein übergeben.
3. **Speicherung**: Der Sub-Baustein speichert den String persistent in der INI-ähnlichen NVS-Struktur unter der Section `SECTION_I1_STORE` und dem Schlüssel `KEY_I1_STORE`.
4. **Rückmeldung**: Nach Abschluss der Speicherung wird ein Ereignis ausgelöst (z. B. `CNF`), das den erfolgreichen Abschluss signalisiert.

Die Übung enthält keine weiteren Verbindungen oder Ereignisverknüpfungen – die gesamte Logik ist im Typ `INI_IN_AND_STORE_AIS` gekapselt. Dadurch wird eine klare Trennung zwischen Applikationslogik und Speicherzugriff erreicht und die Wiederverwendbarkeit erhöht.

Zusammenfassung

Uebung_012k_AIS zeigt, wie man einen sicheren, persistenten Speichervorgang für einen String mithilfe eines AIS-Adapters realisiert. Der Lernende erhält Einblicke in:

- die Verwendung von NVS-Konstanten (`KEY`, `SECTION`),
- die Einbindung eines Adapter-basierten Subbausteins,
- die Parameterübergabe über die 4diac-IDE.

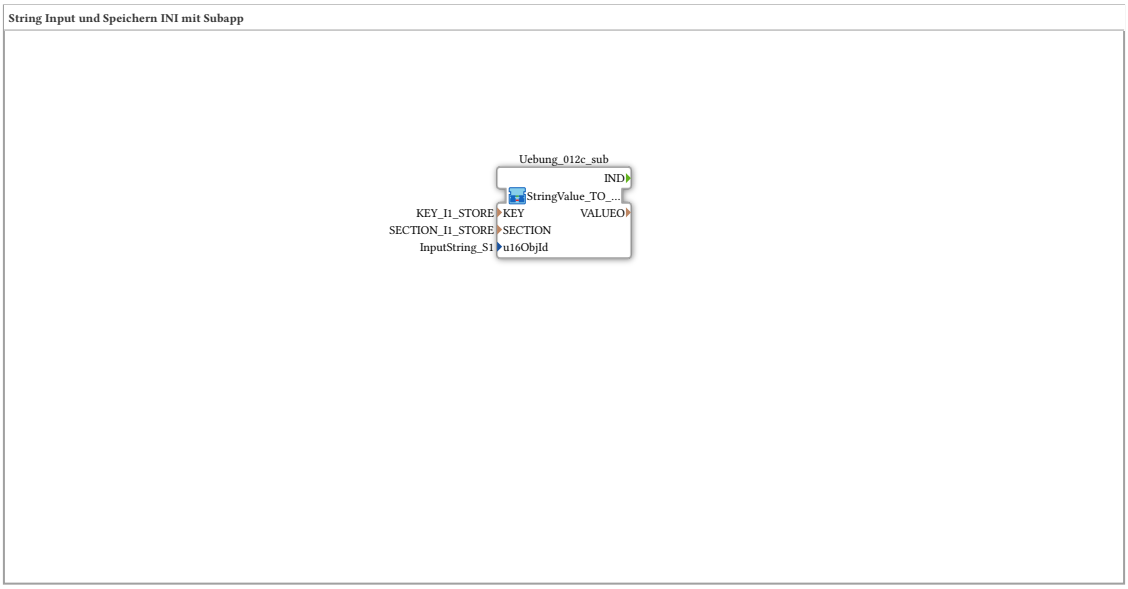
Nach Abschluss der Übung kann der Entwickler dieses Muster auf eigene Anwendungen übertragen, bei denen Benutzereingaben oder Konfigurationsdaten dauerhaft gespeichert werden müssen.

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

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

2.1.2.209 Uebung_012k_AX: String Input und Speichern INI mit Subapp

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_012k_AX (String Input und Speichern INI mit Subapp).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **String Input und Speichern INI mit Subapp**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_012k_AX.SUB, welche die folgende Bausteinstruktur verwendet:

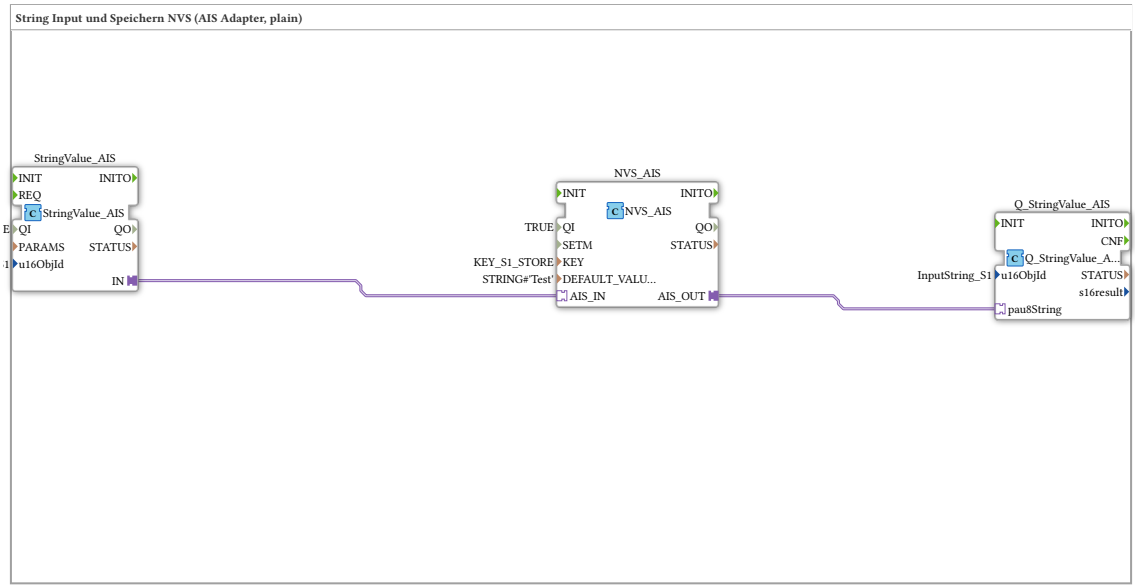
Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Signalweiterleitung**: Die Bausteine verarbeiten Daten- und Steuersignale gemäß ihrer Konfiguration.

Zusammenfassung

Die Übung Uebung_012k_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.210 Uebung_012l_AIS: String Input und Speichern NVS (AIS Adapter, plain)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert die Verwendung des AIS-Adapter-Protokolls zur Kommunikation zwischen einem String-Eingabebaustein und einem nichtflüchtigen Speicher (NVS). Der eingegebene String wird über einen AIS-Adapter an den NVS-Baustein übergeben und dort gespeichert. Ein Lesebaustein stellt den aktuell gespeicherten Wert wieder bereit. Die Übung dient als einfaches Beispiel für die Speicherung von Konfigurations- oder Zustandsdaten mittels des AIS-Modells in 4diac.

Verwendete Funktionsbausteine (FBs)

StringValue_AIS

- **Typ**: `isobus::UT::io::StringValue::StringValue_AIS`
- **Parameter**:
  - `QI` = `TRUE` (Eingang freigeschaltet)
  - `u16ObjId` = `InputString_S1` (Objekt-ID des Eingabestrings)
- **Funktionsweise**: Dieser Baustein stellt einen AIS-Adapter bereit, über den ein String eingegeben werden kann. Er ist die Quelle des zu speichernden Wertes.

NVS_AIS

- **Typ**: `logiBUS::storage::esp32_nvs::NVS_AIS`

- **Parameter**:
  - `QI` = `TRUE` (Eingang freigeschaltet)
  - `KEY` = `KEY_S1_STORE` (Speicherschlüssel im NVS)
  - `DEFAULT_VALUE` = `STRING#'Test'` (Standardwert, falls der Schlüssel nicht existiert)
- **Funktionsweise**: Der Baustein realisiert einen nichtflüchtigen Speicher (NVS) mit AIS-Schnittstelle. Über den AIS_IN wird ein String empfangen und unter dem angegebenen Schlüssel gespeichert. Über den AIS_OUT wird der gespeicherte String (oder der Default-Wert) ausgegeben.

Q_StringValue_AIS

- **Typ**: `isobus::UT::Q::Q_StringValue_AIS`
- **Parameter**:
  - `u16ObjId` = `InputString_S1` (Objekt-ID des Ausgabestrings)
- **Funktionsweise**: Dieser Baustein empfängt über seinen AIS-Adapter den aktuell gespeicherten String vom NVS und stellt ihn für die weitere Verarbeitung (z. B. Anzeige) zur Verfügung.

Programmablauf und Verbindungen

Die drei Funktionsbausteine sind über AIS-Adapter verbunden:

1. **StringValue_AIS.IN → NVS_AIS.AIS_IN**: Der vom Benutzer eingegebene String wird direkt an den NVS-Baustein weitergeleitet.
2. **NVS_AIS.AIS_OUT → Q_StringValue_AIS.pau8String**: Der im NVS gespeicherte (bzw. der Default-) String wird an den Ausgabebaustein gesendet.

Der Ablauf ist zyklisch: Sobald ein neuer String eingegeben wird, aktualisiert der NVS den gespeicherten Wert und gibt diesen über den Ausgang weiter. Die Übung erfordert keine zusätzlichen Ereignisse, da die AIS-Adapter die Datenflüsse selbst steuern.

Lernziele:

- Verständnis des AIS-Adapterkonzepts für den Datenaustausch zwischen Funktionsbausteinen.
- Einfache persistente Speicherung von Strings im NVS.
- Konfiguration von Objekt-IDs und Speicherschlüsseln.

Voraussetzungen: Grundkenntnisse in 4diac und dem AIS-Adaptermodell.

Zusammenfassung

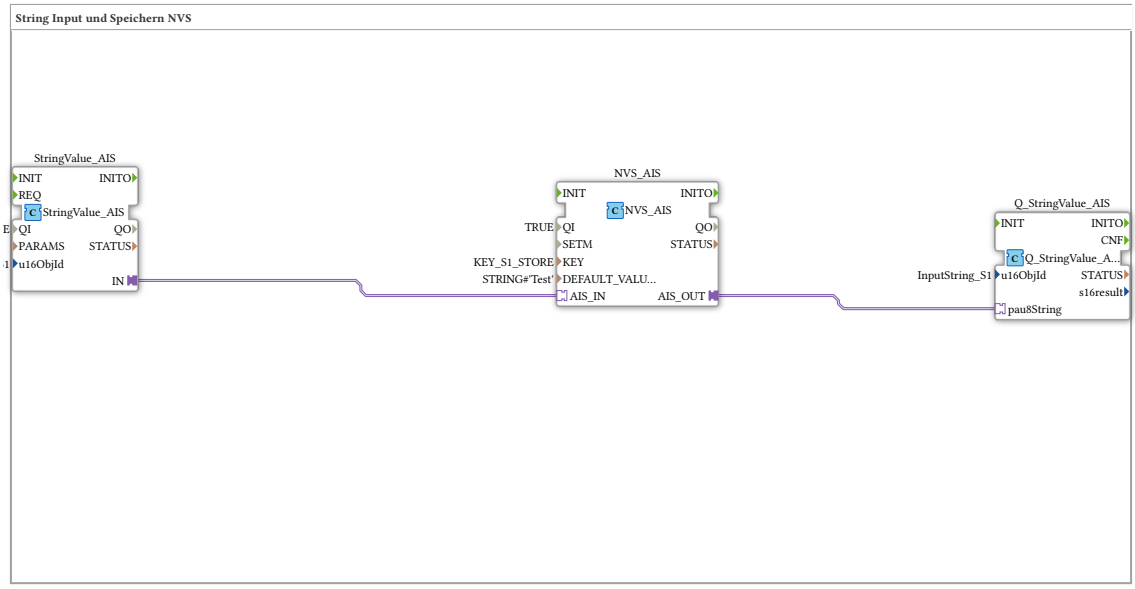
Die Übung **Uebung_012l_AIS** zeigt eine minimalistische Kette: Stringeingabe → NVS-Speicherung → Ausgabe über AIS. Sie verdeutlicht, wie Konfigurationsdaten mit geringem Aufwand dauerhaft gespeichert und wieder ausgelesen werden können. Die Implementierung nutzt das AIS-Adapterprotokoll, ohne dass zusätzliche Ereignisverbindungen nötig sind.

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

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

2.1.2.211 Uebung_012l_AX: String Input und Speichern NVS

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_012l_AX (String Input und Speichern NVS).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **String Input und Speichern NVS**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_012l_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **StringValue_AIS**: Instanz des Typs isobus::UT::io::StringValue::StringValue_AIS.
  - Parameter QI = TRUE
  - Parameter u16ObjId = InputString_S1
- **NVS_AIS**: Instanz des Typs logiBUS::storage::esp32_nvs::NVS_AIS.
  - Parameter QI = TRUE
  - Parameter KEY = KEY_S1_STORE
  - Parameter DEFAULT_VALUE = STRING#‚Test‘
- **Q_StringValue_AIS**: Instanz des Typs isobus::UT::Q::Q_StringValue_AIS.
  - Parameter u16ObjId = InputString_S1

Verbindungen und Schnittstellen

Adapterverbindungen:

- StringValue_AIS.IN -> NVS_AIS.AIS_IN
- NVS_AIS.AIS_OUT -> Q_StringValue_AIS.pau8String

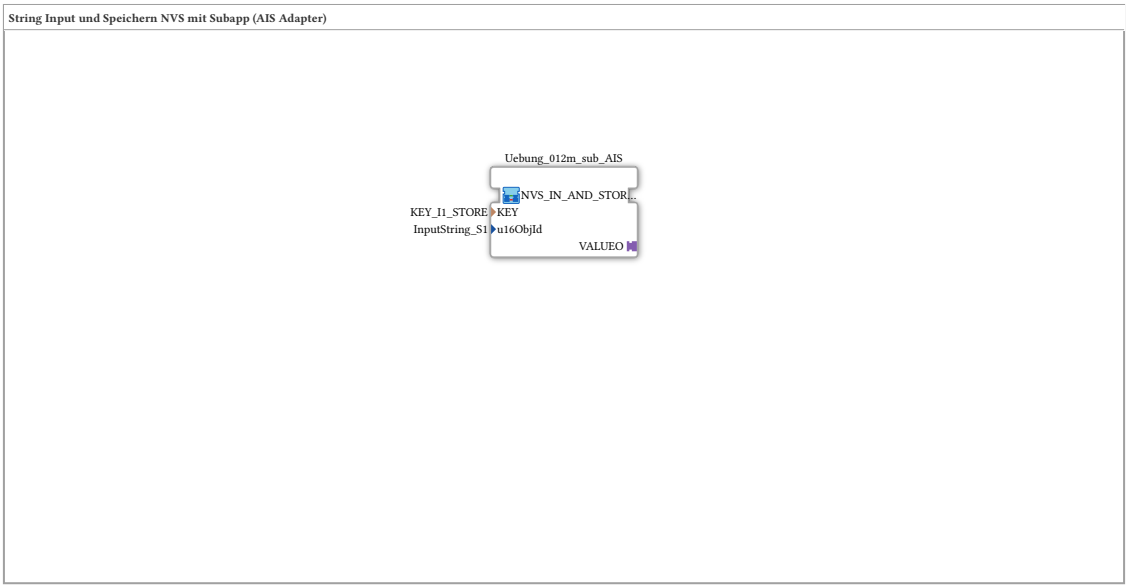
Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_012l_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.212 Uebung_012m_AIS: String Input und Speichern NVS mit Subapp (AIS Adapter)



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung demonstriert die Verwendung eines **AIS-Adapters** (Application Interface Service) zum Einlesen und Speichern eines Strings im **nichtflüchtigen Speicher (NVS)**. Der Funktionsbaustein `NVS_IN_AND_STORE_AIS` wird als SubApp eingesetzt, um einen vorgegebenen Schlüssel (`KEY`) und eine Objekt-ID (`u16ObjId`) zu parametrieren und so einen beliebigen Stringwert persistent zu speichern.

Die Übung zeigt, wie externe Konstanten aus dem Projektimport genutzt werden können, um die Konfiguration sauber zu halten.

Verwendete Funktionsbausteine (FBs)

Die Übung enthält einen einzigen SubApp-Baustein, der die gesamte Funktionalität kapselt.

Sub-Bausteine: `Uebung_012m_sub_AIS`

- **Typ**: `MyLib::sys::NVS_IN_AND_STORE_AIS`
- **Verwendete interne FBs**: Keine (der FB ist als vordefinierter Typ aus einer Bibliothek eingebunden)
- **Parameter**:
  - `KEY` = `KEY_I1_STORE` (Import aus `Uebungen::const::NVS::NVS_Keys::KEY_I1_STORE`)
  - `u16ObjId` = `InputString_S1` (Import aus `Uebungen::const::UT::DefaultPool::InputString_S1`)
- **Funktionsweise**: Der FB `NVS_IN_AND_STORE_AIS` wartet auf einen eingehenden String über den AIS-Adapter. Sobald ein gültiger Wert ankommt, wird dieser unter dem angegebenen NVS-Schlüssel (`KEY`) dauerhaft gespeichert. Die Objekt-ID (`u16ObjId`) identifiziert die Datenquelle oder den Speicherort im DefaultPool. Das Zusammenspiel von AIS und NVS ermöglicht eine hardwarenahe, asynchrone Datenübernahme, ohne dass zyklische Abfragen nötig sind.

Programmablauf und Verbindungen

Die SubApp besitzt keine eigenen Ein-/Ausgangsschnittstellen (InterfaceList ist leer). Sie wird als eigenständige Komponente in eine übergeordnete Applikation eingebettet. Die Verbindung zur Außenwelt erfolgt ausschließlich über den AIS-Adapter, der von der Laufzeitumgebung bereitgestellt wird.

- Die Konstanten `KEY_I1_STORE` und `InputString_S1` müssen im Projekt als Konstanten definiert sein und werden beim Import in die SubApp übernommen.
- Die Ausführung startet, sobald der AIS-Adapter einen neuen String liefert. Der FB speichert diesen Wert dann persistent.

Lernziele:

- Verständnis des AIS-Adapter-Konzepts für asynchrone Datenaufnahme
- Persistente Speicherung von Stringdaten im NVS

- Einsatz von importierten Konstanten zur Parametrierung von Funktionsbausteinen

Schwierigkeitsgrad: Fortgeschritten **Vorkenntnisse**: Grundlagen der 4diac-IDE, NVS-Konzept, Umgang mit Konstanten und Importen

Zusammenfassung

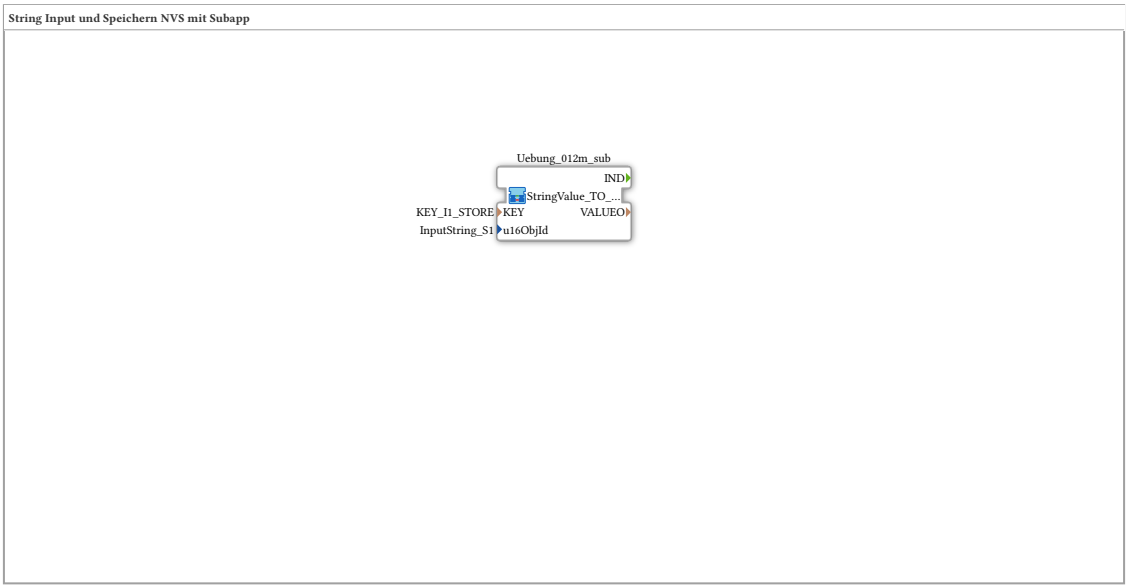
Die Übung `Uebung_012m_AIS` zeigt, wie ein String über einen AIS-Adapter eingelesen und mittels eines vordefinierten Bausteins sicher im nichtflüchtigen Speicher abgelegt wird. Durch die strikte Trennung von Konstanten und Logik bleibt die Konfiguration flexibel und wiederverwendbar.

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

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

2.1.2.213 Uebung_012m_AX: String Input und Speichern NVS mit Subapp

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_012m_AX (String Input und Speichern NVS mit Subapp).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **String Input und Speichern NVS mit Subapp**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_012m_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Signalweiterleitung**: Die Bausteine verarbeiten Daten- und Steuersignale gemäß ihrer Konfiguration.

Zusammenfassung

Die Übung Uebung_012m_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.214 Uebung_012n_AR2: Numeric Value Input PHYS und Speichern NVS via AR2 (bidirektional, direkt)

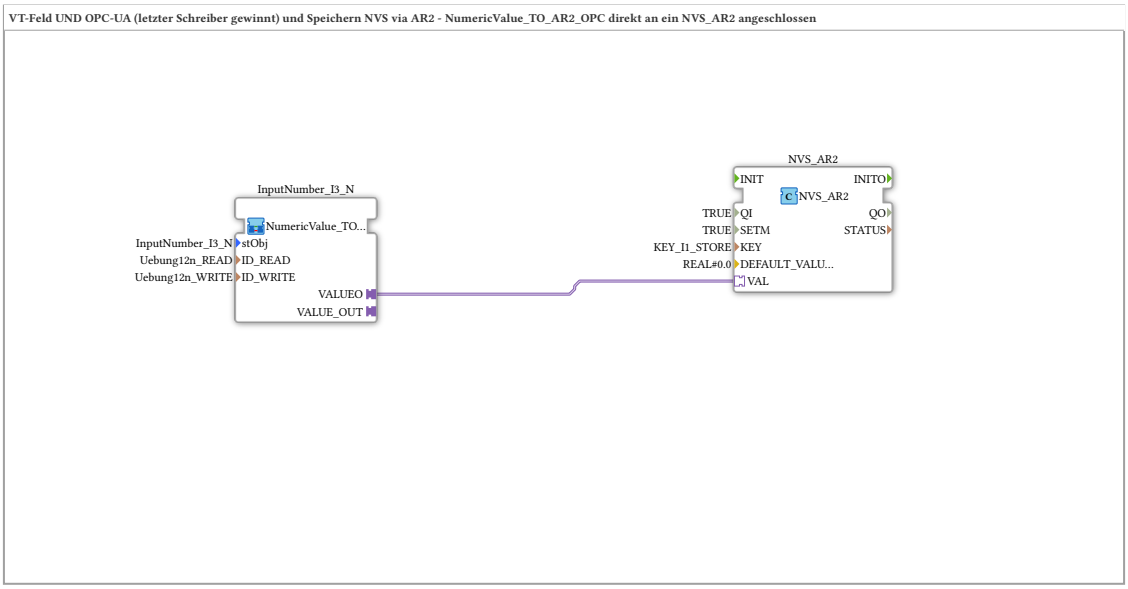
Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_012n_AR2`. Teil einer Serie von Übungen (`Uebung_012d_AR2` bis `Uebung_012o_AR2`), die verschiedene Kombinationen aus Eingabequelle (VT-Feld,

OPC-UA oder beides) und Speicherziel (NVS-Flash oder INI-Datei) über den bidirektionalen AR2-Adapter demonstrieren.

Ziel der Übung

Einen numerischen Wert sowohl über ein VT-Zahlenfeld (NumericValue_TO_AR2_OPC) als auch über OPC-UA - wer zuletzt schreibt, gewinnt (SETM = TRUE) entgegennehmen und dauerhaft in NVS (ESP32-Flash) speichern - unter Verwendung des bidirektionalen AR2-Adapters (direkt verdrahtet, ohne Subapp-Kapselung).

Beschreibung und Komponenten



- **NVS_AR2**: logiBUS::storage::esp32_nvs::NVS_AR2, bidirektionaler AR2-Adapter-Baustein, der den Wert direkt persistiert.

Funktionsweise

Der Zahlenwert wird sowohl über ein VT-Zahlenfeld (NumericValue_TO_AR2_OPC) als auch über OPC-UA - wer zuletzt schreibt, gewinnt (SETM = TRUE) direkt über den bidirektionalen AR2-Adapter an NVS_AR2 weitergegeben, der ihn in NVS (ESP32-Flash) speichert - ohne Zwischenschritt über eine separate Subapp.

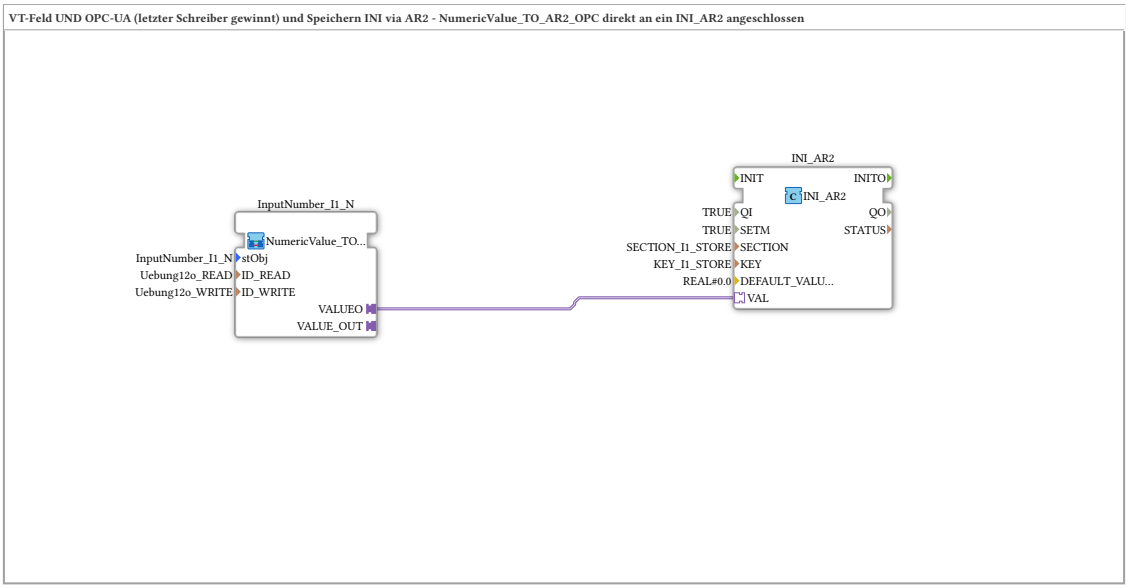
2.1.2.215 Uebung_012o_AR2: Numeric Value Input PHYS und Speichern INI via AR2 (bidirektional, direkt)

Dieser Artikel beschreibt die logiBUS®-Übung Uebung_012o_AR2. Teil einer Serie von Übungen (Uebung_012d_AR2 bis Uebung_012o_AR2), die verschiedene Kombinationen aus Eingabequelle (VT-Feld, OPC-UA oder beides) und Speicherziel (NVS-Flash oder INI-Datei) über den bidirektionalen AR2-Adapter demonstrieren.

Ziel der Übung

Einen numerischen Wert sowohl über ein VT-Zahlenfeld (NumericValue_TO_AR2_OPC) als auch über OPC-UA - wer zuletzt schreibt, gewinnt (SETM = TRUE) entgegennehmen und dauerhaft in INI-Datei speichern - unter Verwendung des bidirektionalen AR2-Adapters (direkt verdrahtet, ohne Subapp-Kapselung).

Beschreibung und Komponenten



- **`INI_AR2`**: `eclipse4diac::storage::INI_AR2`, bidirektionaler AR2-Adapter-Baustein, der den Wert direkt persistiert.

Funktionsweise

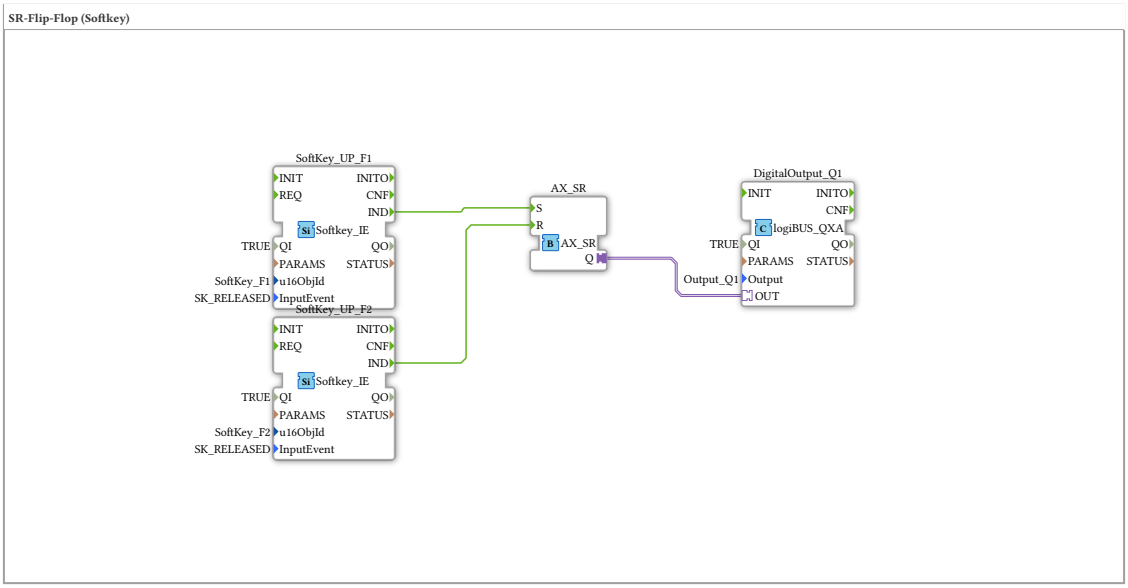
Der Zahlenwert wird sowohl über ein VT-Zahlenfeld (`NumericValue_TO_AR2_OPC`) als auch über OPC-UA - wer zuletzt schreibt, gewinnt (`SETM = TRUE`) direkt über den bidirektionalen `AR2`-Adapter an `INI_AR2` weitergegeben, der ihn in INI-Datei speichert - ohne Zwischenschritt über eine separate Subapp.

2.1.2.216 Uebung_013_AX: SR-Flip-Flop (Softkey)



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_013_AX`.

🎧 Podcast



- ISO 11783-6: Softkeys und das Virtual Terminal verstehen – Dein Schlüssel zur Landmaschinen-Mechatronik

---

Ziel der Übung

Getrennte Ein/Aus Tasten auf dem Touchscreen.

Beschreibung

Die Subapplikation `Uebung_013_AX.SUB` verwendet zwei Softkeys, um ein `AX_SR` Flip-Flop zu steuern.

Funktionsbausteine (FBs)

- **`SoftKey_UP_F1`**: Event `SK_RELEASED` -> Set (`S`).
- **`SoftKey_UP_F2`**: Event `SK_RELEASED` -> Reset (`R`).
- **`AX_SR`**: Speicher.

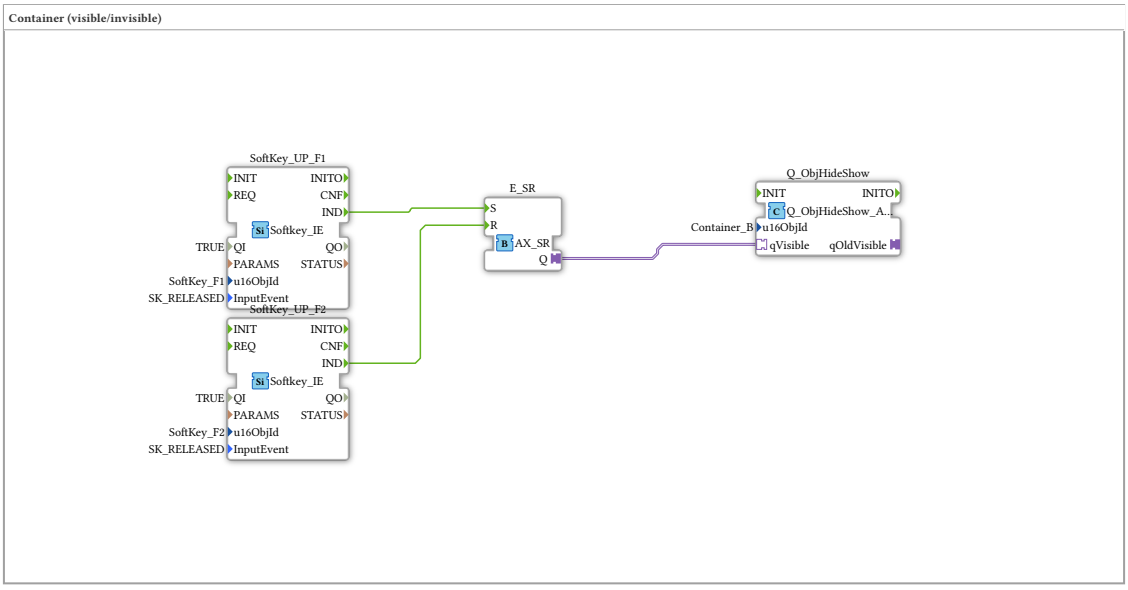
Funktionsweise

- Drücken (und Loslassen) von **F1** schaltet die Funktion ein.
- Drücken (und Loslassen) von **F2** schaltet die Funktion aus.

Dies ist eine klare und sichere Bedienung, oft verwendet für „Start“ und „Stopp“ Symbole.

2.1.2.217 Uebung_014_AX: Container (visible/invisible)

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_014_AX (Container (visible/invisible)).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Container (visible/invisible)**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_014_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **SoftKey_UP_F1**: Instanz des Typs isobus::UT::io::Softkey::Softkey_IE.
  - Parameter QI = TRUE
  - Parameter u16ObjId = SoftKey_F1
  - Parameter InputEvent = SK_RELEASED
- **SoftKey_UP_F2**: Instanz des Typs isobus::UT::io::Softkey::Softkey_IE.
  - Parameter QI = TRUE

  - Parameter u16ObjId = SoftKey_F2
  - Parameter InputEvent = SK_RELEASED
- **E_SR**: Instanz des Typs adapter::events::unidirectional::AX_SR.
- **Q_ObjHideShow**: Instanz des Typs isobus::UT::Q::Q_ObjHideShow_AX.
  - Parameter u16ObjId = Container_B

Verbindungen und Schnittstellen

Adapterverbindungen:

- E_SR.Q -> Q_ObjHideShow.qVisible

Ereignisverbindungen:

- SoftKey_UP_F1.IND -> E_SR.S
- SoftKey_UP_F2.IND -> E_SR.R

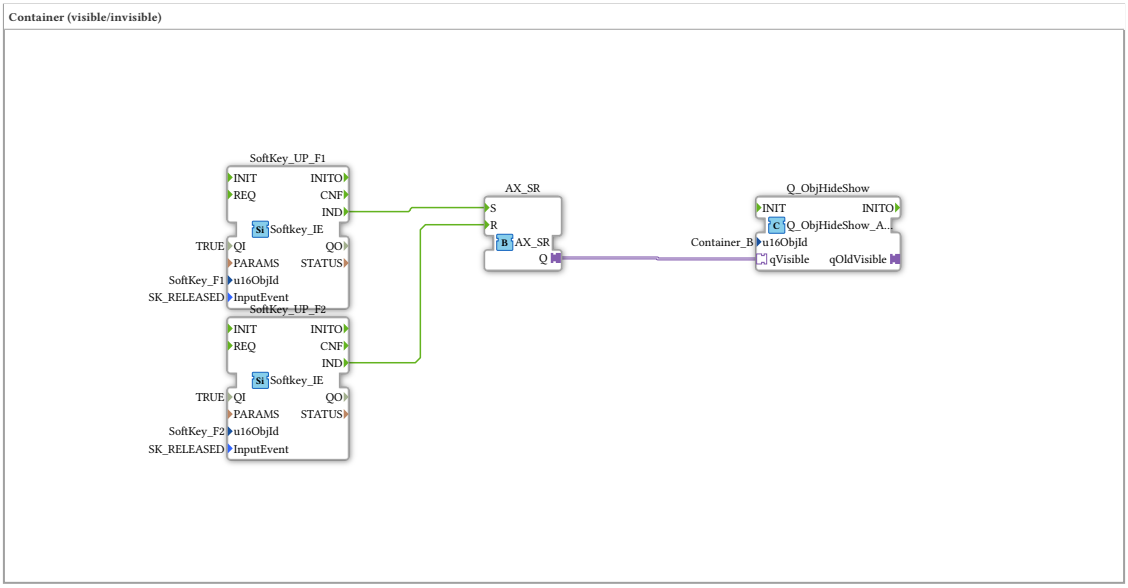
Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_014_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.218 Uebung_014_AXa: Container (visible/invisible)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert die Steuerung der Sichtbarkeit eines Containers (Objekts) durch zwei Softkeys. Mit SoftKey_F1 wird der Container eingeblendet, mit SoftKey_F2 ausgeblendet. Dabei wird ein Set-Reset-Flipflop (AX_SR) verwendet, um den Zustand zu speichern und an einen Hide/Show-Baustein weiterzugeben.

Verwendete Funktionsbausteine (FBs)

SoftKey_UP_F1 (Typ: isobus::UT::io::Softkey::Softkey_IE)

- **Parameter**:

  - `QI` = TRUE
  - `u16ObjId` = SoftKey_F1
  - `InputEvent` = SK_RELEASED
- **Ereignisausgang/-eingang**: Ereignisausgang `IND` (wird beim Loslassen von F1 ausgelöst)
- **Funktionsweise**: Dieser Baustein erfasst das Loslassen des Softkeys F1 und gibt ein Ereignis an seinem Ausgang `IND` aus.

SoftKey_UP_F2 (Typ: isobus::UT::io::Softkey::Softkey_IE)

- **Parameter**:
  - `QI` = TRUE
  - `u16ObjId` = SoftKey_F2
  - `InputEvent` = SK_RELEASED
- **Ereignisausgang/-eingang**: Ereignisausgang `IND`
- **Funktionsweise**: Analog zu SoftKey_UP_F1, jedoch für Softkey F2.

AX_SR (Typ: adapter::events::unidirectional::AX_SR)

- **Parameter**: Keine expliziten Parameter.
- **Adapteranschlüsse**:
  - Eingänge: `S` (Set), `R` (Reset)
  - Ausgang: `Q` (Zustand)
- **Funktionsweise**: Ein Set-Reset-Flipflop. Bei einem Ereignis auf `S` wird der Ausgang `Q` auf TRUE gesetzt, bei einem Ereignis auf `R` wird er zurückgesetzt (FALSE). Der Zustand bleibt erhalten, bis das nächste Ereignis eintrifft.

Q_ObjHideShow (Typ: isobus::UT::Q::Q_ObjHideShow_AX)

- **Parameter**:
  - `u16ObjId` = Container_B
- **Adapteranschluss**: `qVisible` (Adapter-Eingang, erwartet einen booleschen Wert)
- **Funktionsweise**: Dieser Baustein steuert die Sichtbarkeit des Objekts mit der ID `Container_B`. Wenn am Adapter `qVisible` der Wert TRUE anliegt, wird das Objekt sichtbar geschaltet, bei FALSE unsichtbar.

Programmablauf und Verbindungen

Die Ereignisausgänge der Softkey-Bausteine sind wie folgt verbunden:

- Das Ereignis `IND` von **SoftKey_UP_F1** ist mit dem Set-Eingang `S` von **AX_SR** verbunden.
- Das Ereignis `IND` von **SoftKey_UP_F2** ist mit dem Reset-Eingang `R` von **AX_SR** verbunden.

Der Zustandsausgang `Q` von **AX_SR** wird über den Adapter `qVisible` an den Baustein **Q_ObjHideShow** weitergeleitet.

Ablauf:

1. Wenn der Bediener Softkey F1 loslässt, wird ein Set-Ereignis an AX_SR gesendet. Dadurch wird der Ausgang `Q` auf TRUE gesetzt.
2. Der TRUE-Wert aktiviert den Baustein **Q_ObjHideShow**, wodurch der Container `Container_B` sichtbar wird.
3. Beim Loslassen von Softkey F2 wird ein Reset-Ereignis an AX_SR gesendet. `Q` wird auf FALSE gesetzt, der Container wird unsichtbar.

Lernziele:

- Steuerung der Sichtbarkeit eines Objekts über zwei Softkeys.
- Verwendung eines Set-Reset-Flipflops (AX_SR) zur Zustandspeicherung.
- Verbindung von Ereignis- und Datenadaptern in einer Subapplikation.

Schwierigkeitsgrad: Einfach **Benötigte Vorkenntnisse**: Grundlagen der Ereignisverarbeitung und Adapterverbindungen in 4diac.

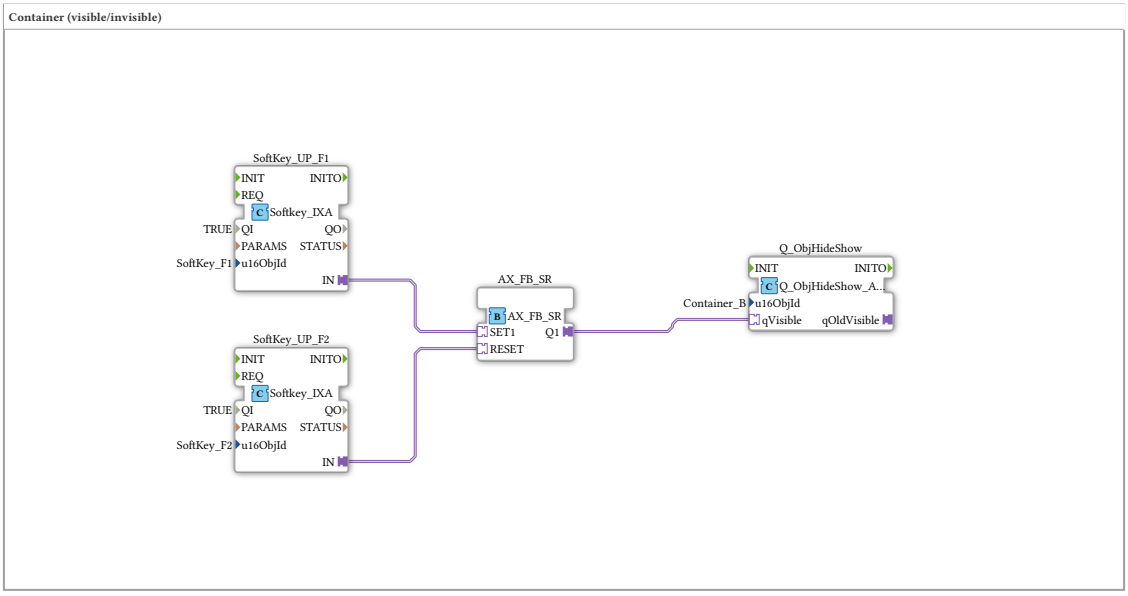
Zusammenfassung

Die Übung zeigt eine typische Anwendung zur Ein-/Ausblendung eines Containers mithilfe zweier Softkeys. Der Set-Reset-Baustein speichert den gewünschten Sichtbarkeitszustand, der über einen Adapter an den Hide/Show-Baustein übergeben wird. Dies ist ein einfaches Beispiel für die Zustandssteuerung in der industriellen Automatisierung mit 4diac.

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

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

2.1.2.219 Uebung_014_AXb: Container (visible/invisible)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert die Nutzung von Softkeys in Kombination mit einem SR-Flipflop, um ein grafisches Objekt (Container_B) ein- und auszublenden. Der Funktionsbaustein überwacht die Tastendrücke der Softkeys F1 (Setzen) und F2 (Rücksetzen) und steuert über ein SR-Glied die Sichtbarkeit des Containers. Die zugehörigen Konstanten *Container_B*, *SoftKey_F1* und *SoftKey_F2* sind aus einem globalen Pool importiert.

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: Uebung_014_AXb

Die Übung besteht aus einer Subapplikation, die folgende interne Funktionsbausteine enthält:

- **SoftKey_UP_F1**: `isobus::UT::io::Softkey::Softkey_IXA`
  - Parameter: `QI` = `TRUE`, `u16ObjId` = `SoftKey_F1`
  - Ereignisausgang: `IN` (bei Tastendruck)
  - Funktionsweise: Überwacht den Softkey F1 und gibt ein Ereignis am Ausgang `IN` aus, sobald die Taste gedrückt wird.
- **SoftKey_UP_F2**: `isobus::UT::io::Softkey::Softkey_IXA`
  - Parameter: `QI` = `TRUE`, `u16ObjId` = `SoftKey_F2`
  - Funktionsweise: Analog zu SoftKey_UP_F1, jedoch für die Taste F2.
- **AX_FB_SR**: `adapter::iec61131::bistableElements::AX_FB_SR`
  - Parameter: Keine
  - Eingänge: `SET1` (Ereignis), `RESET` (Ereignis)
  - Ausgänge: `Q1` (Wert)
  - Funktionsweise: Ein SR-Flipflop (bistabiles Element). Bei einem Ereignis auf `SET1` wird der Ausgang `Q1` auf TRUE gesetzt; bei `RESET` wird er auf FALSE zurückgesetzt.
- **Q_ObjHideShow**: `isobus::UT::Q::Q_ObjHideShow_AX`
  - Parameter: `u16ObjId` = `Container_B`
  - Eingang: `qVisible` (Wert)

– Funktionsweise: Steuert die Sichtbarkeit des durch `u16ObjId` referenzierten Objekts (Container_B). Ist der Eingang `qVisible` = TRUE, wird der Container angezeigt; bei FALSE wird er ausgeblendet.

Programmablauf und Verbindungen

Die Verbindungen innerhalb der Subapplikation sind wie folgt realisiert:

1. **SoftKey_UP_F1** – Drücken der Taste F1 ⟶ Ereignisausgang `IN` wird aktiv.
2. **SoftKey_UP_F2** – Drücken der Taste F2 ⟶ Ereignisausgang `IN` wird aktiv.
3. **AX_FB_SR** – Der Ereigniseingang `SET1` ist mit dem Ausgang von SoftKey_UP_F1 verbunden. Der Ereigniseingang `RESET` ist mit dem Ausgang von SoftKey_UP_F2 verbunden.
4. **Q_ObjHideShow** – Der Wert-Eingang `qVisible` ist mit dem Ausgang `Q1` des SR-Flipflops verbunden.

Ablauf:

- Ein Druck auf die Softkey-Taste **F1** setzt das SR-Flipflop, der Ausgang `Q1` wird = TRUE ⟶ der Container wird angezeigt.
- Ein Druck auf die Softkey-Taste **F2** setzt das Flipflop zurück, `Q1` wird = FALSE ⟶ der Container wird ausgeblendet.

Die Übung erfordert keine weiteren Datentypen oder Parameter. Der Benutzer muss lediglich die beiden Softkey-Tasten bedienen, um das Ein- und Ausblenden des Containers zu steuern.

Zusammenfassung

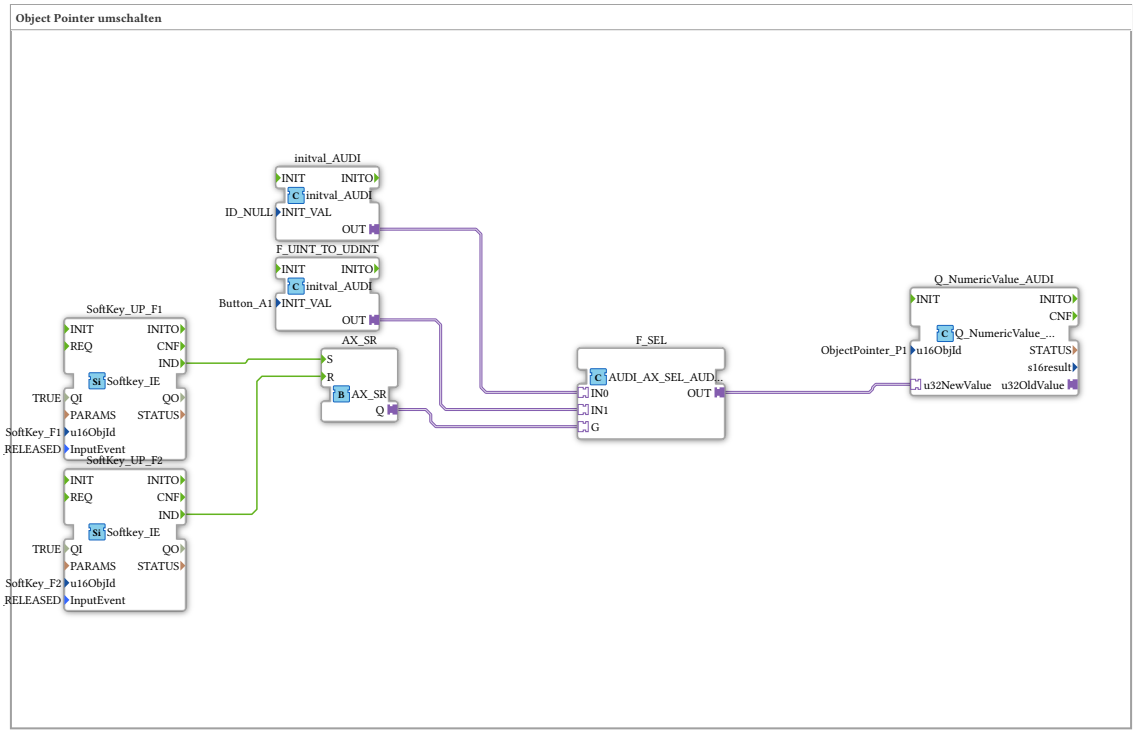
In dieser Übung wird ein einfaches, aber praxisnahes Steuerungsmuster für die Visualisierung in einem ISOBUS-Terminal umgesetzt. Durch die Kombination von Softkey-Ereignissen, einem SR-Flipflop und einem Sichtbarkeitsbaustein wird das Verhalten eines „Ein/Aus"-Schalters für ein grafisches Objekt realisiert. Die Übung verdeutlicht das Zusammenspiel von Ereignis- und Datenflüssen in einer 4diac-Subapplikation und die Nutzung vordefinierter Bibliotheksbausteine für die ISOBUS-Kommunikation.

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

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

2.1.2.220 Uebung_015_AX: Object Pointer umschalten

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_015_AX (Object Pointer umschalten).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Object Pointer umschalten**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_015_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **SoftKey_UP_F1**: Instanz des Typs isobus::UT::io::Softkey::Softkey_IE.
  - Parameter QI = TRUE
  - Parameter u16ObjId = SoftKey_F1
  - Parameter InputEvent = SK_RELEASED
- **SoftKey_UP_F2**: Instanz des Typs isobus::UT::io::Softkey::Softkey_IE.
  - Parameter QI = TRUE
  - Parameter u16ObjId = SoftKey_F2
  - Parameter InputEvent = SK_RELEASED
- **AX_SR**: Instanz des Typs adapter::events::unidirectional::AX_SR.
- **F_UINT_TO_UDINT**: Instanz des Typs adapter::types::unidirectional::AUDI::initval::initval_AUDI.
  - Parameter INIT_VAL = Button_A1
- **F_SEL**: Instanz des Typs adapter::iec61131::selection::AUDI_AX_SEL_AUDI.
- **Q_NumericValue_AUDI**: Instanz des Typs isobus::UT::Q::Q_NumericValue_AUDI.
  - Parameter u16ObjId = ObjectPointer_P1
- **initval_AUDI**: Instanz des Typs adapter::types::unidirectional::AUDI::initval::initval_AUDI.
  - Parameter INIT_VAL = ID_NULL

Verbindungen und Schnittstellen

Adapterverbindungen:

- F_UINT_TO_UDINT.OUT -> F_SEL.IN1
- AX_SR.Q -> F_SEL.G
- initval_AUDI.OUT -> F_SEL.IN0
- F_SEL.OUT -> Q_NumericValue_AUDI.u32NewValue

Ereignisverbindungen:

- SoftKey_UP_F1.IND -> AX_SR.S
- SoftKey_UP_F2.IND -> AX_SR.R

Programmablauf und Funktionsweise

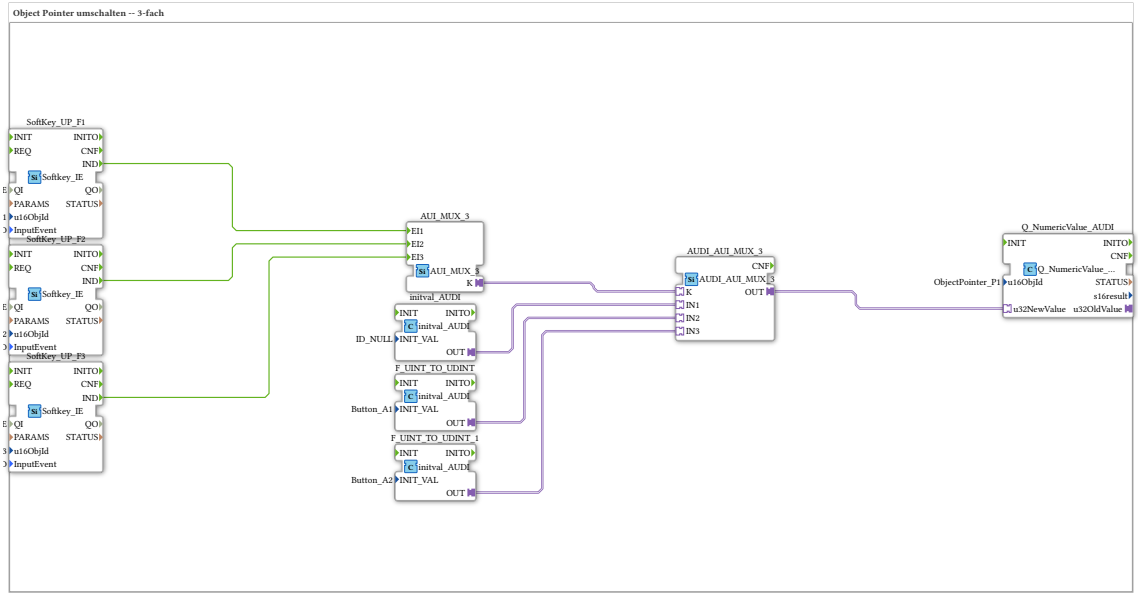
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_015_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.221 Uebung_015a_AX: Object Pointer umschalten – 3-fach

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_015a_AX (Object Pointer umschalten – 3-fach).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Object Pointer umschalten – 3-fach**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_015a_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **SoftKey_UP_F1**: Instanz des Typs isobus::UT::io::Softkey::Softkey_IE.
  - Parameter QI = TRUE
  - Parameter u16ObjId = SoftKey_F1
  - Parameter InputEvent = SK_RELEASED
- **SoftKey_UP_F2**: Instanz des Typs isobus::UT::io::Softkey::Softkey_IE.
  - Parameter QI = TRUE
  - Parameter u16ObjId = SoftKey_F2
  - Parameter InputEvent = SK_RELEASED
- **F_UINT_TO_UDINT**: Instanz des Typs adapter::types::unidirectional::AUDI::initval::initval_AUDI.
  - Parameter INIT_VAL = Button_A1
- **Q_NumericValue_AUDI**: Instanz des Typs isobus::UT::Q::Q_NumericValue_AUDI.
  - Parameter u16ObjId = ObjectPointer_P1
- **SoftKey_UP_F3**: Instanz des Typs isobus::UT::io::Softkey::Softkey_IE.
  - Parameter QI = TRUE
  - Parameter u16ObjId = SoftKey_F3
  - Parameter InputEvent = SK_RELEASED
- **F_UINT_TO_UDINT_1**: Instanz des Typs adapter::types::unidirectional::AUDI::initval::initval_AUDI.
  - Parameter INIT_VAL = Button_A2
- **initval_AUDI**: Instanz des Typs adapter::types::unidirectional::AUDI::initval::initval_AUDI.
  - Parameter INIT_VAL = ID_NULL
- **AUDI_AUI_MUX_3**: Instanz des Typs adapter::selection::unidirectional::AUDI_AUI_MUX_3.
- **AUI_MUX_3**: Instanz des Typs adapter::events::unidirectional::AUI_MUX_3.

Verbindungen und Schnittstellen

Adapterverbindungen:

- AUDI_AUI_MUX_3.OUT -> Q_NumericValue_AUDI.u32NewValue
- F_UINT_TO_UDINT_1.OUT -> AUDI_AUI_MUX_3.IN3
- F_UINT_TO_UDINT.OUT -> AUDI_AUI_MUX_3.IN2
- initval_AUDI.OUT -> AUDI_AUI_MUX_3.IN1

- AUI_MUX_3.K -> AUDI_AUI_MUX_3.K

Ereignisverbindungen:

- SoftKey_UP_F1.IND -> AUI_MUX_3.EI1
- SoftKey_UP_F2.IND -> AUI_MUX_3.EI2
- SoftKey_UP_F3.IND -> AUI_MUX_3.EI3

Programmablauf und Funktionsweise

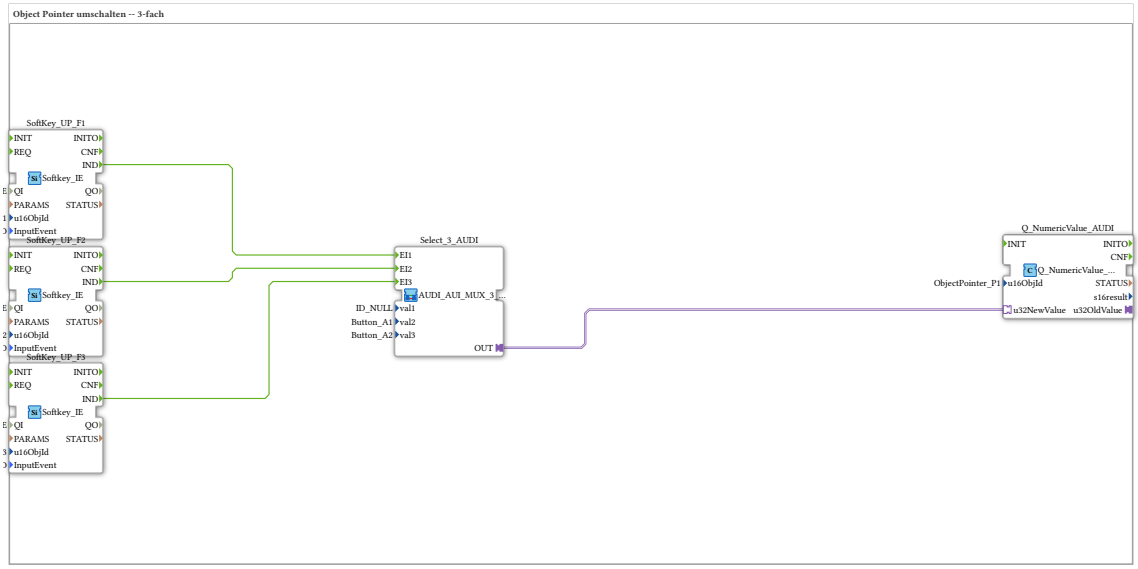
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_015a_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.222 Uebung_015b_AX: Object Pointer umschalten – 3-fach

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_015b_AX (Object Pointer umschalten – 3-fach).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Object Pointer umschalten – 3-fach**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_015b_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **SoftKey_UP_F1**: Instanz des Typs isobus::UT::io::Softkey::Softkey_IE.
  - Parameter QI = TRUE
  - Parameter u16ObjId = SoftKey_F1
  - Parameter InputEvent = SK_RELEASED
- **SoftKey_UP_F2**: Instanz des Typs isobus::UT::io::Softkey::Softkey_IE.
  - Parameter QI = TRUE
  - Parameter u16ObjId = SoftKey_F2
  - Parameter InputEvent = SK_RELEASED

- **SoftKey_UP_F3**: Instanz des Typs isobus::UT::io::Softkey::Softkey_IE.
  - Parameter QI = TRUE
  - Parameter u16ObjId = SoftKey_F3
  - Parameter InputEvent = SK_RELEASED
- **Q_NumericValue_AUDI**: Instanz des Typs isobus::UT::Q::Q_NumericValue_AUDI.
  - Parameter u16ObjId = ObjectPointer_P1

Verbindungen und Schnittstellen

Adapterverbindungen:

- Select_3_AUDI.OUT -> Q_NumericValue_AUDI.u32NewValue

Ereignisverbindungen:

- SoftKey_UP_F1.IND -> Select_3_AUDI.EI1
- SoftKey_UP_F2.IND -> Select_3_AUDI.EI2
- SoftKey_UP_F3.IND -> Select_3_AUDI.EI3

Programmablauf und Funktionsweise

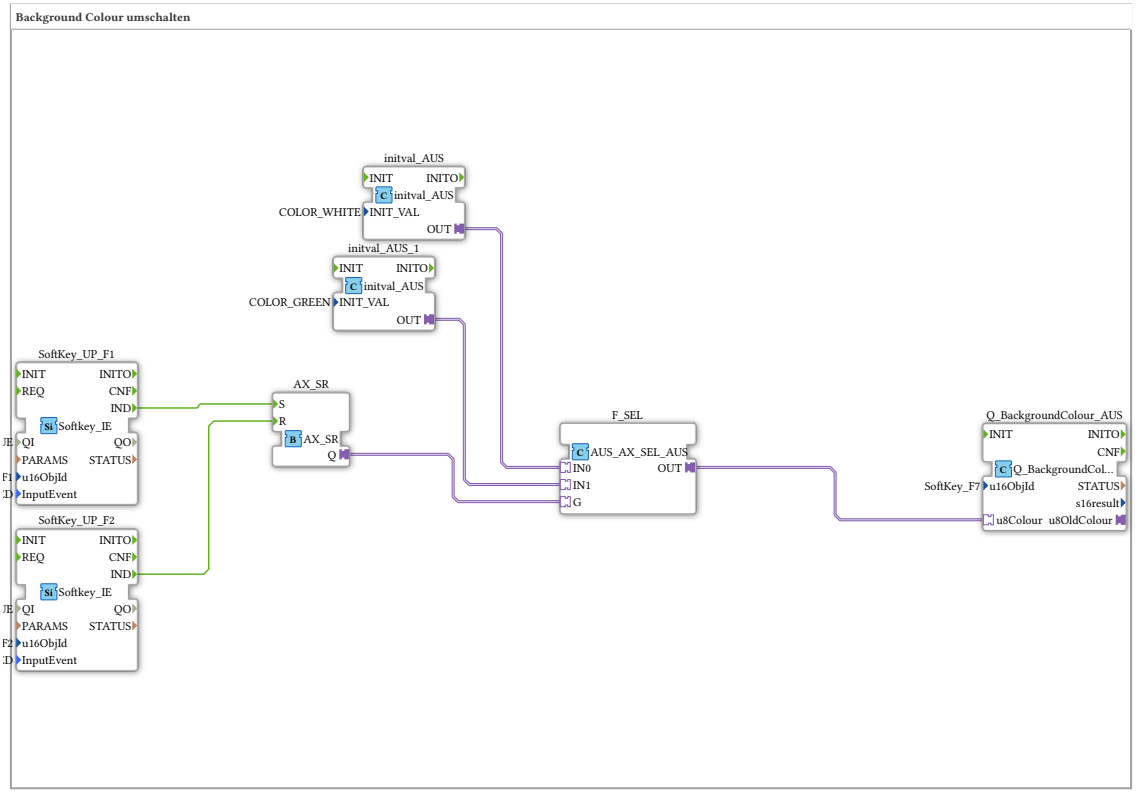
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_015b_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.223 Uebung_016_AX: Background Colour umschalten

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_016_AX (Background Colour umschalten).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Background Colour umschalten**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_016_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **SoftKey_UP_F1**: Instanz des Typs isobus::UT::io::Softkey::Softkey_IE.
  - Parameter QI = TRUE
  - Parameter u16ObjId = SoftKey_F1
  - Parameter InputEvent = SK_RELEASED
- **SoftKey_UP_F2**: Instanz des Typs isobus::UT::io::Softkey::Softkey_IE.
  - Parameter QI = TRUE
  - Parameter u16ObjId = SoftKey_F2
  - Parameter InputEvent = SK_RELEASED
- **AX_SR**: Instanz des Typs adapter::events::unidirectional::AX_SR.
- **F_SEL**: Instanz des Typs adapter::iec61131::selection::AUS_AX_SEL_AUS.
- **Q_BackgroundColour_AUS**: Instanz des Typs isobus::UT::Q::Q_BackgroundColour_AUS.
  - Parameter u16ObjId = SoftKey_F7
- **initval_AUS**: Instanz des Typs adapter::types::unidirectional::AUS::initval::initval_AUS.
  - Parameter INIT_VAL = COLOR_WHITE
- **initval_AUS_1**: Instanz des Typs adapter::types::unidirectional::AUS::initval::initval_AUS.
  - Parameter INIT_VAL = COLOR_GREEN

Verbindungen und Schnittstellen

Adapterverbindungen:

- AX_SR.Q -> F_SEL.G
- F_SEL.OUT -> Q_BackgroundColour_AUS.u8Colour
- initval_AUS_1.OUT -> F_SEL.IN1
- initval_AUS.OUT -> F_SEL.IN0

Ereignisverbindungen:

- SoftKey_UP_F1.IND -> AX_SR.S
- SoftKey_UP_F2.IND -> AX_SR.R

Programmablauf und Funktionsweise

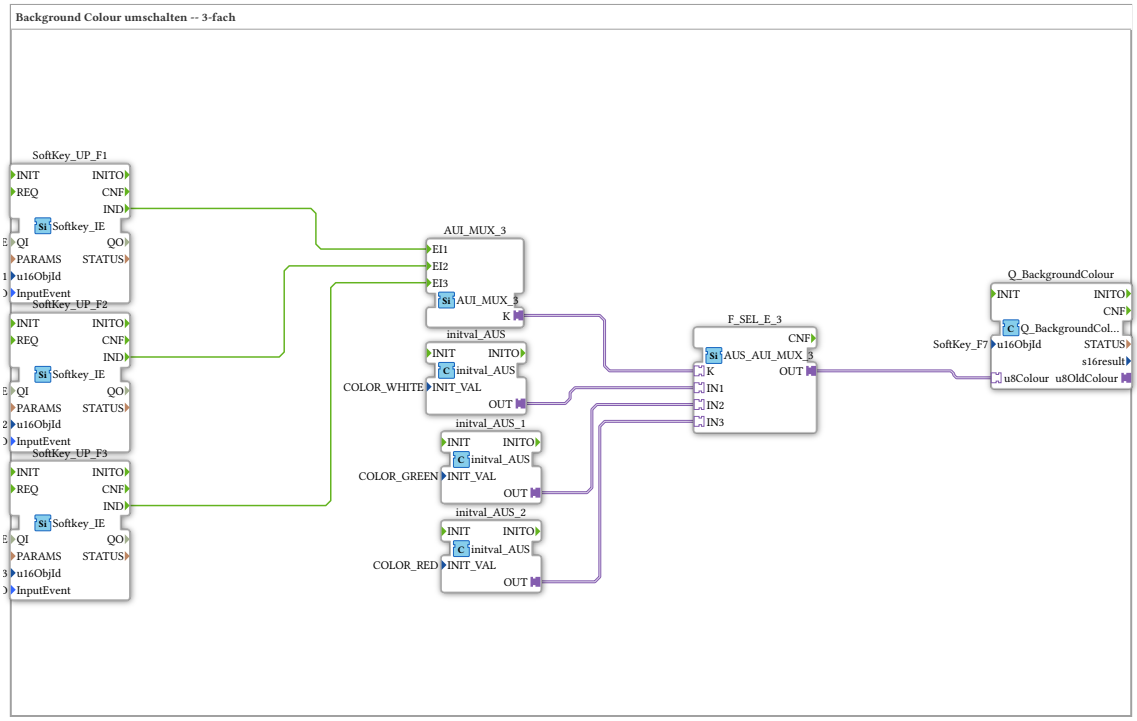
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_016_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.224 Uebung_016a_AX: Background Colour umschalten – 3-fach

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_016a_AX (Background Colour umschalten – 3-fach).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Background Colour umschalten – 3-fach**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_016a_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **SoftKey_UP_F1**: Instanz des Typs isobus::UT::io::Softkey::Softkey_IE.
  - Parameter QI = TRUE
  - Parameter u16ObjId = SoftKey_F1
  - Parameter InputEvent = SK_RELEASED
- **SoftKey_UP_F2**: Instanz des Typs isobus::UT::io::Softkey::Softkey_IE.
  - Parameter QI = TRUE
  - Parameter u16ObjId = SoftKey_F2
  - Parameter InputEvent = SK_RELEASED
- **SoftKey_UP_F3**: Instanz des Typs isobus::UT::io::Softkey::Softkey_IE.
  - Parameter QI = TRUE
  - Parameter u16ObjId = SoftKey_F3
  - Parameter InputEvent = SK_RELEASED
- **F_SEL_E_3**: Instanz des Typs adapter::selection::unidirectional::AUS_AUI_MUX_3.
- **Q_BackgroundColour**: Instanz des Typs isobus::UT::Q::Q_BackgroundColour_AUS.
  - Parameter u16ObjId = SoftKey_F7
- **initval_AUS**: Instanz des Typs adapter::types::unidirectional::AUS::initval::initval_AUS.
  - Parameter INIT_VAL = COLOR_WHITE
- **initval_AUS_1**: Instanz des Typs adapter::types::unidirectional::AUS::initval::initval_AUS.
  - Parameter INIT_VAL = COLOR_GREEN
- **initval_AUS_2**: Instanz des Typs adapter::types::unidirectional::AUS::initval::initval_AUS.
  - Parameter INIT_VAL = COLOR_RED
- **AUI_MUX_3**: Instanz des Typs adapter::events::unidirectional::AUI_MUX_3.

Verbindungen und Schnittstellen

Adapterverbindungen:

- F_SEL_E_3.OUT -> Q_BackgroundColour.u8Colour
- initval_AUS.OUT -> F_SEL_E_3.IN1
- initval_AUS_1.OUT -> F_SEL_E_3.IN2
- initval_AUS_2.OUT -> F_SEL_E_3.IN3
- AUI_MUX_3.K -> F_SEL_E_3.K

Ereignisverbindungen:

- SoftKey_UP_F1.IND -> AUI_MUX_3.EI1
- SoftKey_UP_F2.IND -> AUI_MUX_3.EI2
- SoftKey_UP_F3.IND -> AUI_MUX_3.EI3

Programmablauf und Funktionsweise

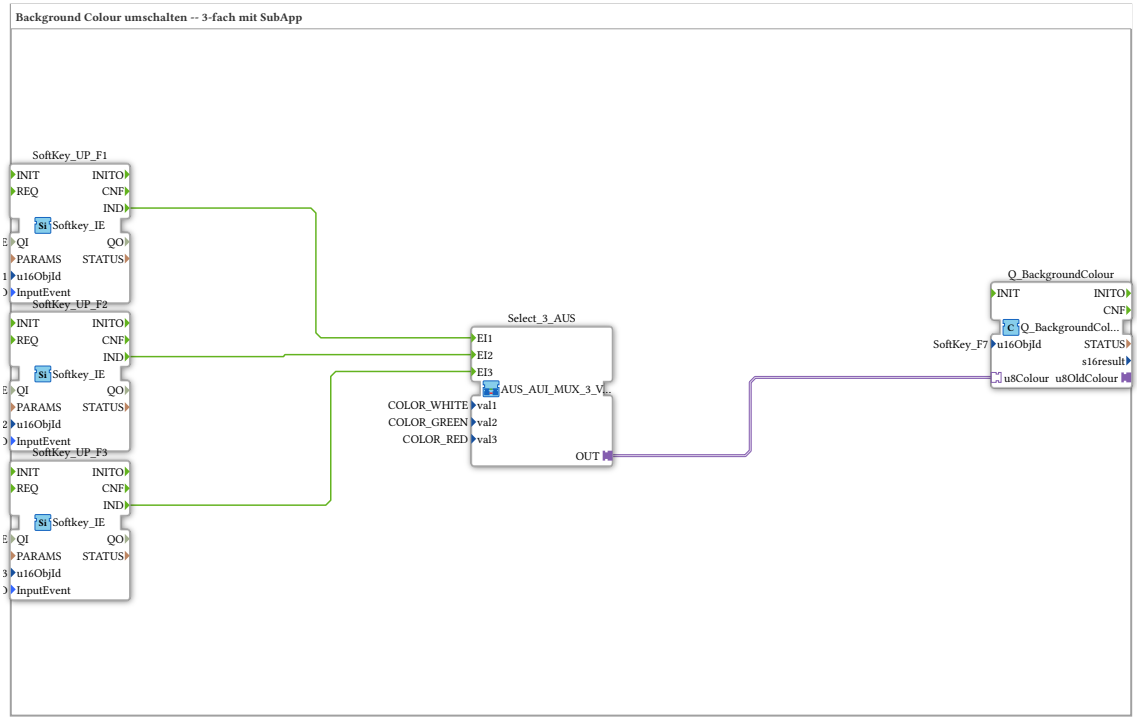
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_016a_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.225 Uebung_016b_AX: Background Colour umschalten – 3-fach mit SubApp

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_016b_AX (Background Colour umschalten – 3-fach mit SubApp).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Background Colour umschalten – 3-fach mit SubApp**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_016b_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **SoftKey_UP_F1**: Instanz des Typs isobus::UT::io::Softkey::Softkey_IE.
  - Parameter QI = TRUE
  - Parameter u16ObjId = SoftKey_F1
  - Parameter InputEvent = SK_RELEASED
- **SoftKey_UP_F2**: Instanz des Typs isobus::UT::io::Softkey::Softkey_IE.
  - Parameter QI = TRUE
  - Parameter u16ObjId = SoftKey_F2
  - Parameter InputEvent = SK_RELEASED
- **SoftKey_UP_F3**: Instanz des Typs isobus::UT::io::Softkey::Softkey_IE.
  - Parameter QI = TRUE
  - Parameter u16ObjId = SoftKey_F3
  - Parameter InputEvent = SK_RELEASED
- **Q_BackgroundColour**: Instanz des Typs isobus::UT::Q::Q_BackgroundColour_AUS.
  - Parameter u16ObjId = SoftKey_F7

Verbindungen und Schnittstellen

Adapterverbindungen:

- Select_3_AUS.OUT -> Q_BackgroundColour.u8Colour

Ereignisverbindungen:

- SoftKey_UP_F1.IND -> Select_3_AUS.EI1
- SoftKey_UP_F2.IND -> Select_3_AUS.EI2
- SoftKey_UP_F3.IND -> Select_3_AUS.EI3

Programmablauf und Funktionsweise

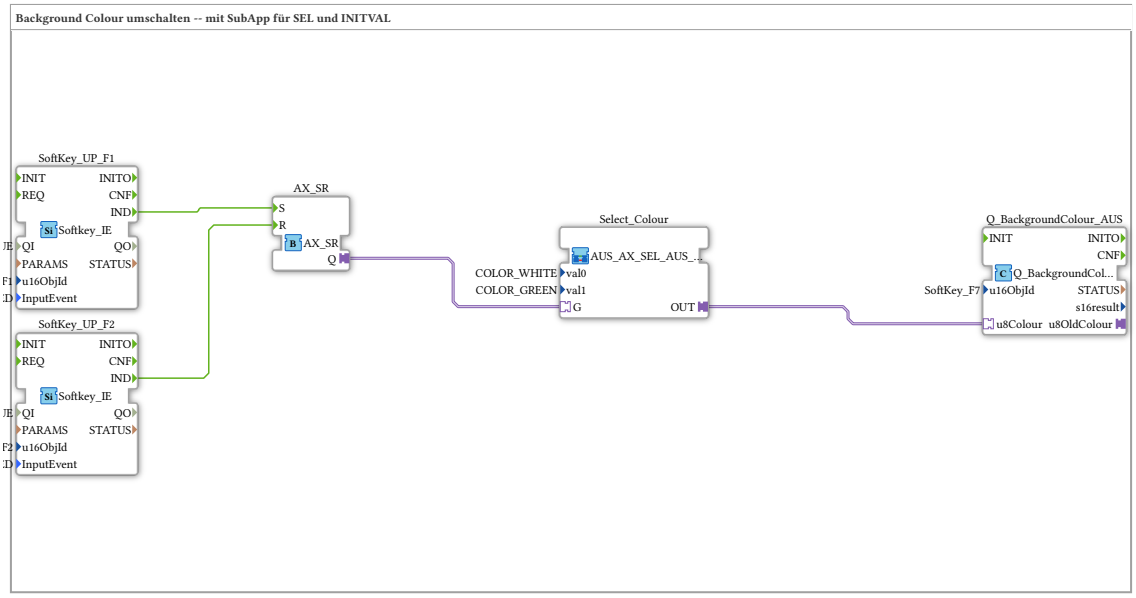
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_016b_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.226 Uebung_016c_AX: Background Colour umschalten – mit SubApp für SEL und INITVAL

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_016c_AX (Background Colour umschalten – mit SubApp für SEL und INITVAL).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Background Colour umschalten – mit SubApp für SEL und INITVAL**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_016c_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **SoftKey_UP_F1**: Instanz des Typs isobus::UT::io::Softkey::Softkey_IE.
  - Parameter QI = TRUE
  - Parameter u16ObjId = SoftKey_F1
  - Parameter InputEvent = SK_RELEASED
- **SoftKey_UP_F2**: Instanz des Typs isobus::UT::io::Softkey::Softkey_IE.
  - Parameter QI = TRUE
  - Parameter u16ObjId = SoftKey_F2
  - Parameter InputEvent = SK_RELEASED
- **AX_SR**: Instanz des Typs adapter::events::unidirectional::AX_SR.
- **Q_BackgroundColour_AUS**: Instanz des Typs isobus::UT::Q::Q_BackgroundColour_AUS.
  - Parameter u16ObjId = SoftKey_F7

Verbindungen und Schnittstellen

Adapterverbindungen:

- AX_SR.Q -> Select_Colour.G
- Select_Colour.OUT -> Q_BackgroundColour_AUS.u8Colour

Ereignisverbindungen:

- SoftKey_UP_F1.IND -> AX_SR.S
- SoftKey_UP_F2.IND -> AX_SR.R

Programmablauf und Funktionsweise

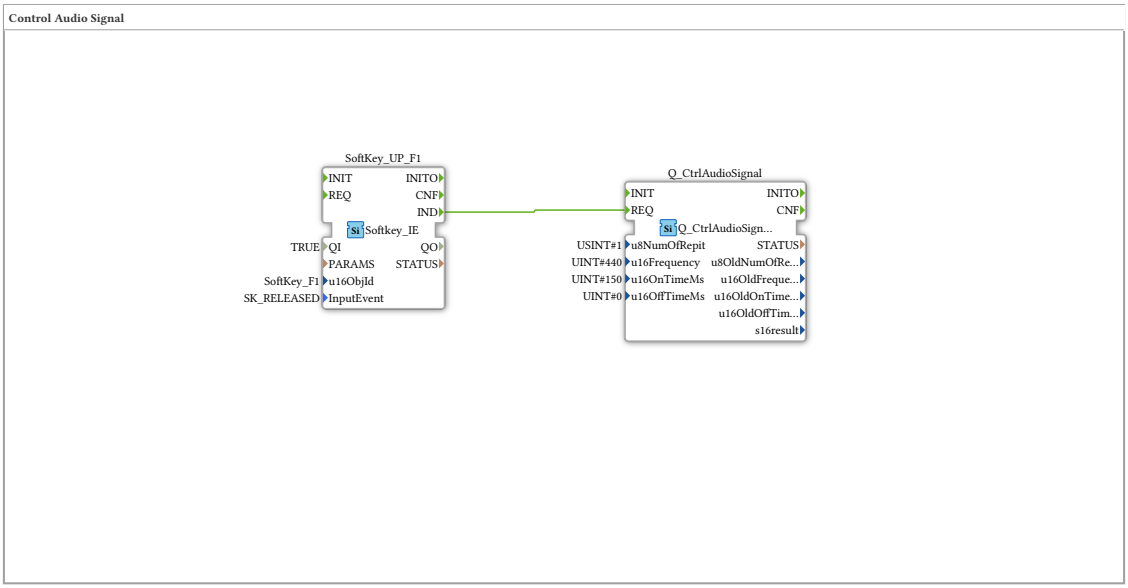
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_016c_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.227 Uebung_017_AX: Control Audio Signal

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_017_AX (Control Audio Signal).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Control Audio Signal**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_017_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **SoftKey_UP_F1**: Instanz des Typs isobus::UT::io::Softkey::Softkey_IE.
  - Parameter QI = TRUE
  - Parameter u16ObjId = SoftKey_F1
  - Parameter InputEvent = SK_RELEASED
- **Q_CtrlAudioSignal**: Instanz des Typs isobus::UT::Q::Q_CtrlAudioSignal.
  - Parameter u8NumOfRepit = 1
  - Parameter u16Frequency = 440
  - Parameter u16OnTimeMs = 150
  - Parameter u16OffTimeMs = 0

Verbindungen und Schnittstellen

Ereignisverbindungen:

- SoftKey_UP_F1.IND -> Q_CtrlAudioSignal.REQ

Programmablauf und Funktionsweise

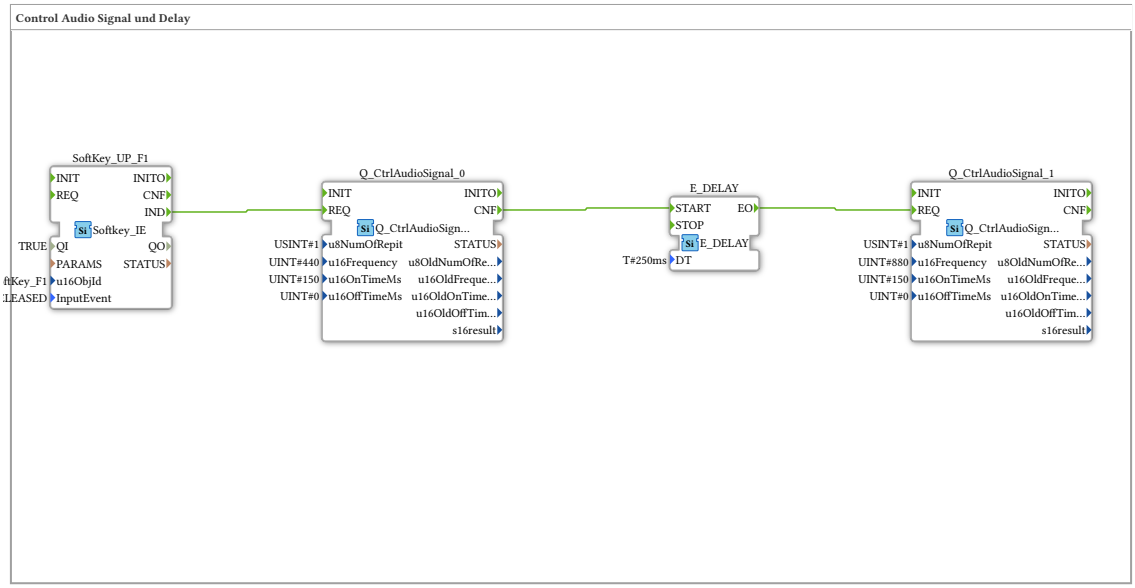
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_017_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.228 Uebung_018_AX: Control Audio Signal und Delay

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_018_AX (Control Audio Signal und Delay).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Control Audio Signal und Delay**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_018_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **SoftKey_UP_F1**: Instanz des Typs isobus::UT::io::Softkey::Softkey_IE.
  - Parameter QI = TRUE
  - Parameter u16ObjId = SoftKey_F1
  - Parameter InputEvent = SK_RELEASED
- **Q_CtrlAudioSignal_0**: Instanz des Typs isobus::UT::Q::Q_CtrlAudioSignal.
  - Parameter u8NumOfRepit = 1
  - Parameter u16Frequency = 440
  - Parameter u16OnTimeMs = 150
  - Parameter u16OffTimeMs = 0
- **Q_CtrlAudioSignal_1**: Instanz des Typs isobus::UT::Q::Q_CtrlAudioSignal.
  - Parameter u8NumOfRepit = 1
  - Parameter u16Frequency = 880
  - Parameter u16OnTimeMs = 150
  - Parameter u16OffTimeMs = 0
- **E_DELAY**: Instanz des Typs iec61499::events::E_DELAY.
  - Parameter DT = T#250ms

Verbindungen und Schnittstellen

Ereignisverbindungen:

- SoftKey_UP_F1.IND -> Q_CtrlAudioSignal_0.REQ
- E_DELAY.EO -> Q_CtrlAudioSignal_1.REQ
- Q_CtrlAudioSignal_0.CNF -> E_DELAY.START

Programmablauf und Funktionsweise

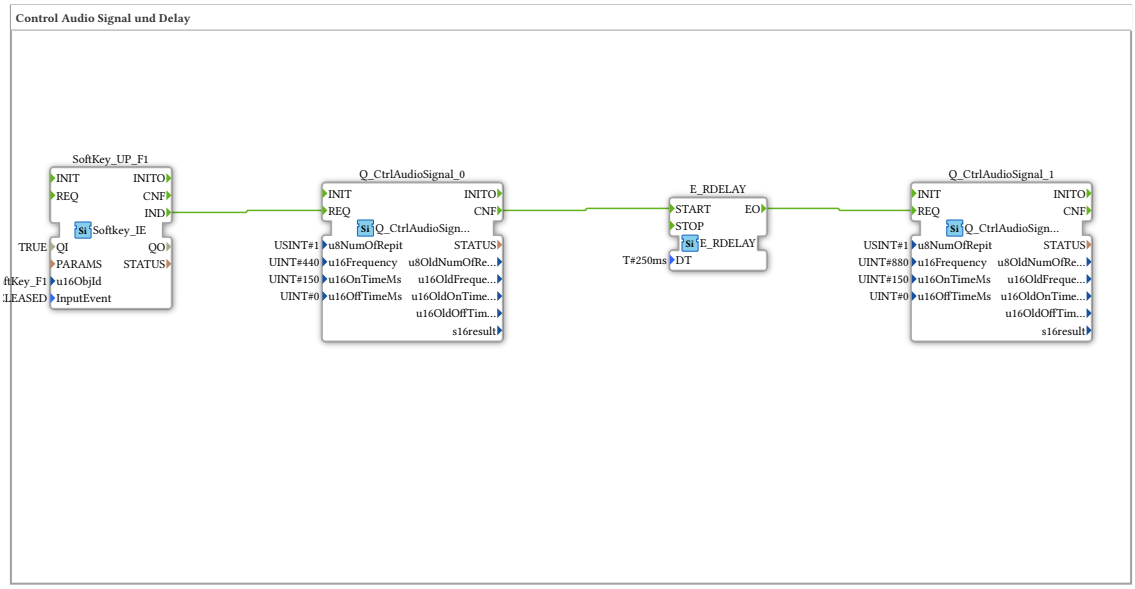
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_018_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.229 Uebung_018a_AX: Control Audio Signal und Delay

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_018a_AX (Control Audio Signal und Delay).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Control Audio Signal und Delay**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_018a_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **SoftKey_UP_F1**: Instanz des Typs isobus::UT::io::Softkey::Softkey_IE.
  - Parameter QI = TRUE
  - Parameter u16ObjId = SoftKey_F1
  - Parameter InputEvent = SK_RELEASED
- **Q_CtrlAudioSignal_0**: Instanz des Typs isobus::UT::Q::Q_CtrlAudioSignal.
  - Parameter u8NumOfRepit = 1
  - Parameter u16Frequency = 440
  - Parameter u16OnTimeMs = 150
  - Parameter u16OffTimeMs = 0
- **Q_CtrlAudioSignal_1**: Instanz des Typs isobus::UT::Q::Q_CtrlAudioSignal.
  - Parameter u8NumOfRepit = 1
  - Parameter u16Frequency = 880
  - Parameter u16OnTimeMs = 150
  - Parameter u16OffTimeMs = 0
- **E_RDELAY**: Instanz des Typs iec61499::events::E_RDELAY.

- Parameter DT = T#250ms

Verbindungen und Schnittstellen

Ereignisverbindungen:

- SoftKey_UP_F1.IND -> Q_CtrlAudioSignal_0.REQ
- E_RDELAY.EO -> Q_CtrlAudioSignal_1.REQ
- Q_CtrlAudioSignal_0.CNF -> E_RDELAY.START

Programmablauf und Funktionsweise

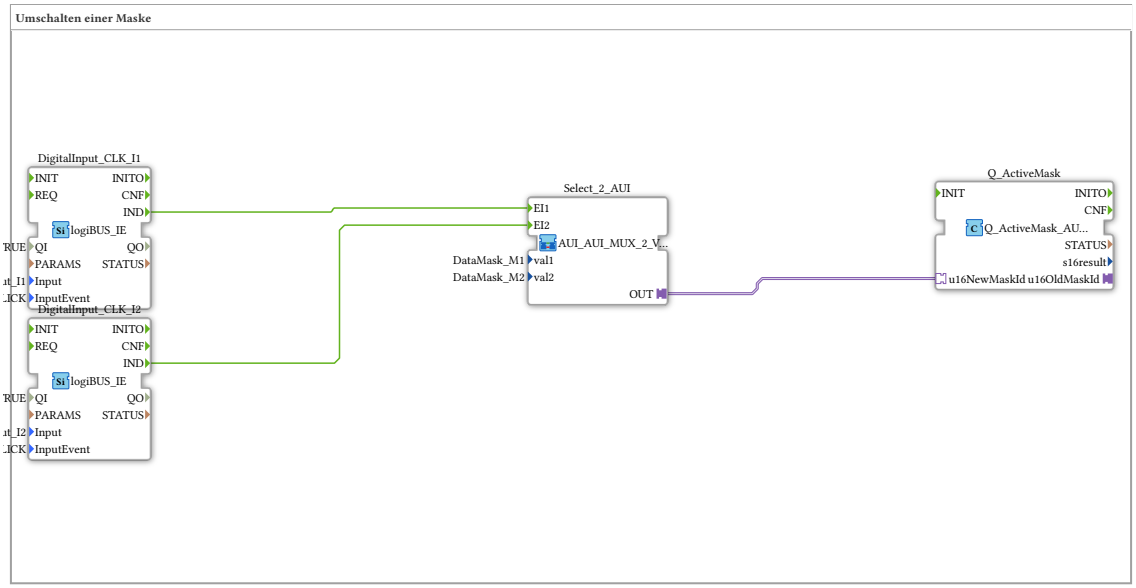
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_018a_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.230 Uebung_019_AX: Umschalten einer Maske

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_019_AX (Umschalten einer Maske).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Umschalten einer Maske**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_019_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_CLK_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **DigitalInput_CLK_I2**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I2

  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **Q_ActiveMask**: Instanz des Typs isobus::UT::Q::Q_ActiveMask_AUI.

Verbindungen und Schnittstellen

Adapterverbindungen:

- Select_2_AUI.OUT -> Q_ActiveMask.u16NewMaskId

Ereignisverbindungen:

- DigitalInput_CLK_I1.IND -> Select_2_AUI.EI1
- DigitalInput_CLK_I2.IND -> Select_2_AUI.EI2

Programmablauf und Funktionsweise

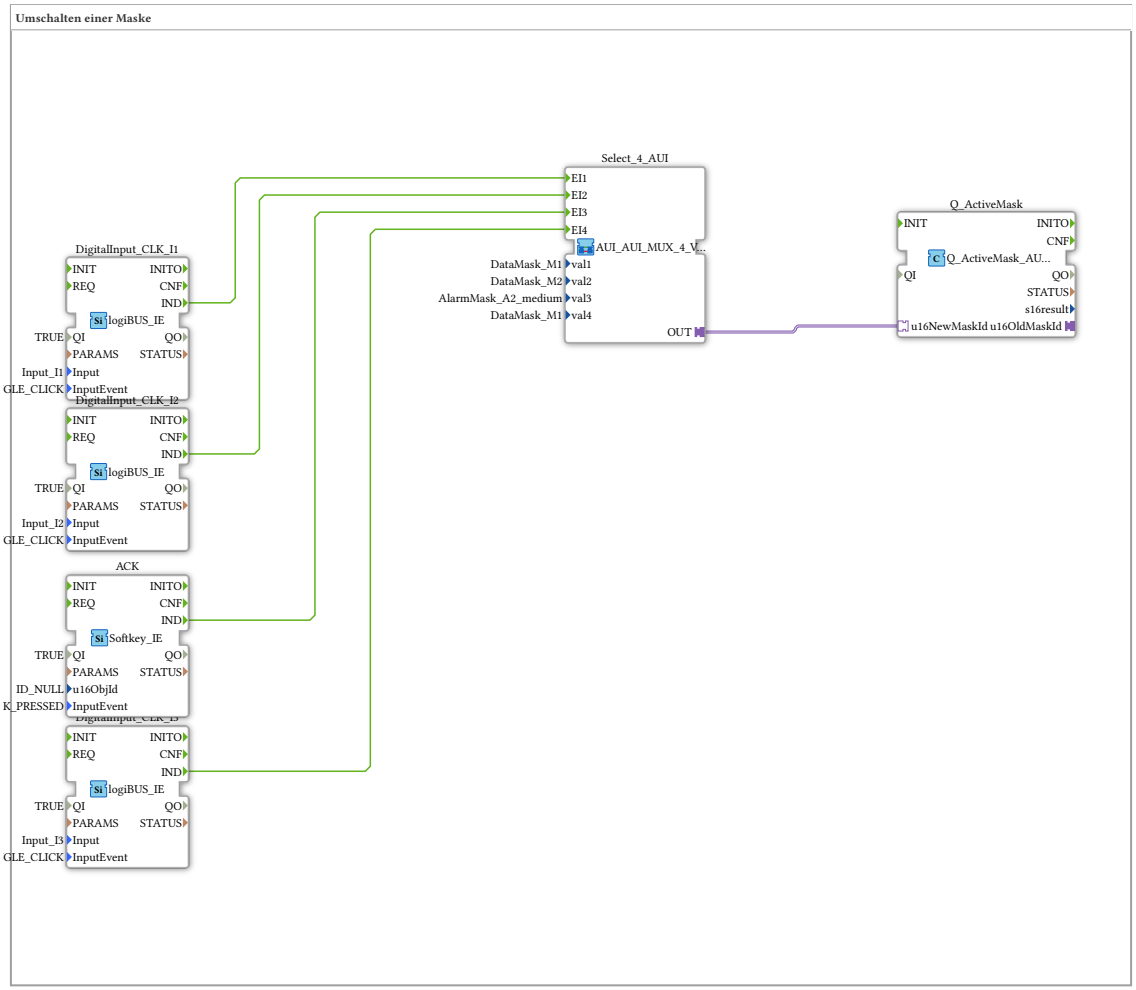
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_019_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.231 Uebung_019a_AX: Umschalten einer Maske

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_019a_AX (Umschalten einer Maske).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Umschalten einer Maske**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_019a_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_CLK_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **DigitalInput_CLK_I2**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I2
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **DigitalInput_CLK_I3**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I3
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **ACK**: Instanz des Typs isobus::UT::io::Softkey::Softkey_IE.
  - Parameter QI = TRUE
  - Parameter u16ObjId = ID_NULL
  - Parameter InputEvent = SK_PRESSED
- **Q_ActiveMask**: Instanz des Typs isobus::UT::Q::Q_ActiveMask_AUI.

Verbindungen und Schnittstellen

Adapterverbindungen:

- Select_4_AUI.OUT -> Q_ActiveMask.u16NewMaskId

Ereignisverbindungen:

- DigitalInput_CLK_I1.IND -> Select_4_AUI.EI1
- DigitalInput_CLK_I2.IND -> Select_4_AUI.EI2
- DigitalInput_CLK_I3.IND -> Select_4_AUI.EI4
- ACK.IND -> Select_4_AUI.EI3

Programmablauf und Funktionsweise

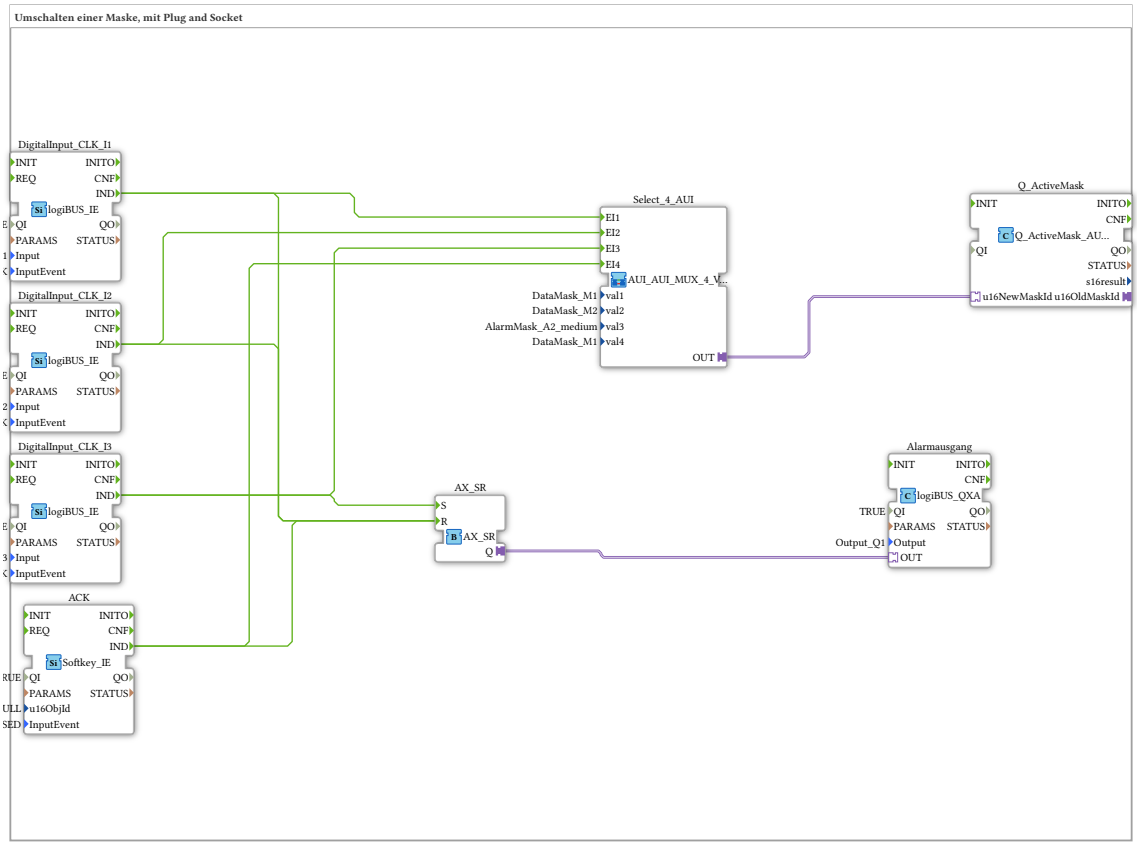
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_019a_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.232 Uebung_019b_AX: Umschalten einer Maske, mit Plug and Socket

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_019b_AX (Umschalten einer Maske, mit Plug and Socket).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Umschalten einer Maske, mit Plug and Socket**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_019b_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_CLK_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **DigitalInput_CLK_I2**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I2
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **DigitalInput_CLK_I3**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I3
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **ACK**: Instanz des Typs isobus::UT::io::Softkey::Softkey_IE.
  - Parameter QI = TRUE
  - Parameter u16ObjId = ID_NULL
  - Parameter InputEvent = SK_PRESSED
- **Q_ActiveMask**: Instanz des Typs isobus::UT::Q::Q_ActiveMask_AUI.
- **AX_SR**: Instanz des Typs adapter::events::unidirectional::AX_SR.
- **Alarmausgang**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1

Verbindungen und Schnittstellen

Adapterverbindungen:

- Select_4_AUI.OUT -> Q_ActiveMask.u16NewMaskId
- AX_SR.Q -> Alarmausgang.OUT

Ereignisverbindungen:

- DigitalInput_CLK_I1.IND -> Select_4_AUI.EI1
- DigitalInput_CLK_I2.IND -> Select_4_AUI.EI2
- DigitalInput_CLK_I3.IND -> Select_4_AUI.EI3
- ACK.IND -> Select_4_AUI.EI4
- DigitalInput_CLK_I3.IND -> AX_SR.S
- DigitalInput_CLK_I2.IND -> AX_SR.R
- DigitalInput_CLK_I1.IND -> AX_SR.R
- ACK.IND -> AX_SR.R

Programmablauf und Funktionsweise

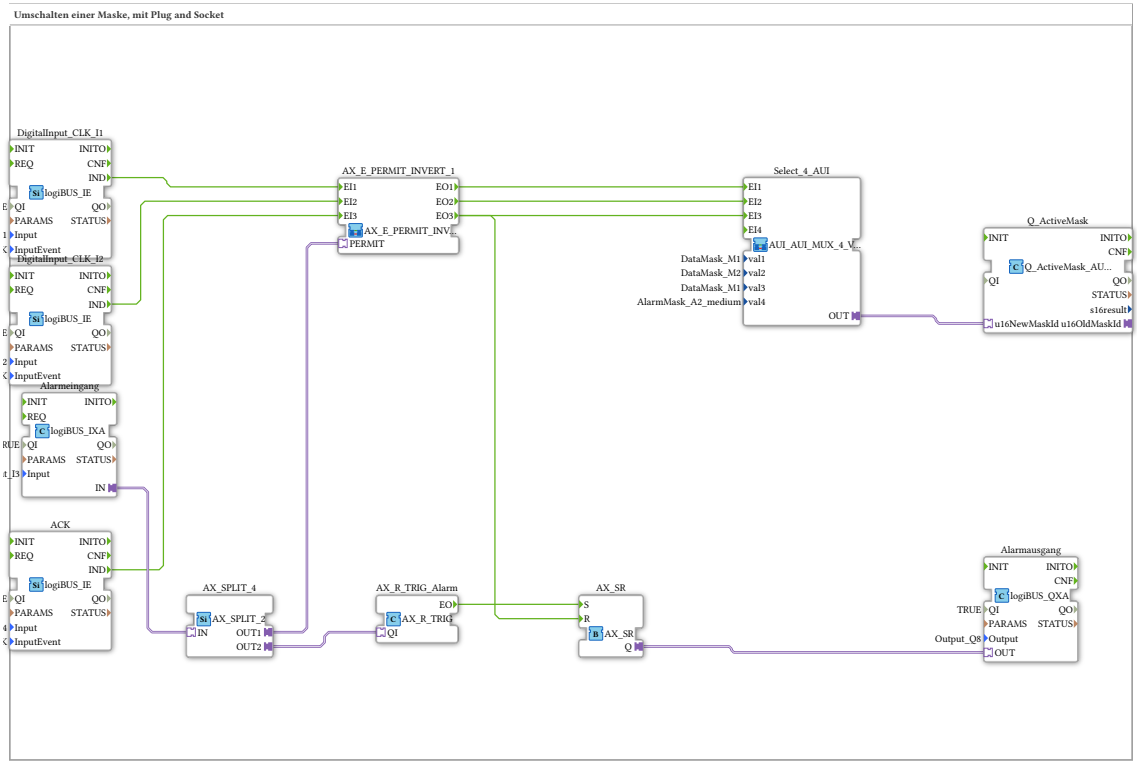
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_019b_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.233 Uebung_019c_AX: Umschalten einer Maske, mit Plug and Socket

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_019c_AX (Umschalten einer Maske, mit Plug and Socket).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Umschalten einer Maske, mit Plug and Socket**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_019c_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_CLK_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **DigitalInput_CLK_I2**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I2
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **Q_ActiveMask**: Instanz des Typs isobus::UT::Q::Q_ActiveMask_AUI.
- **ACK**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I4
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **AX_SR**: Instanz des Typs adapter::events::unidirectional::AX_SR.
- **Alarmausgang**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q8
- **Alarmeingang**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I3
- **AX_R_TRIG_Alarm**: Instanz des Typs adapter::events::unidirectional::AX_R_TRIG.
- **AX_SPLIT_4**: Instanz des Typs adapter::events::unidirectional::AX_SPLIT_2.

Verbindungen und Schnittstellen

Adapterverbindungen:

- Select_4_AUI.OUT -> Q_ActiveMask.u16NewMaskId
- AX_SR.Q -> Alarmausgang.OUT
- Alarmeingang.IN -> AX_SPLIT_4.IN
- AX_SPLIT_4.OUT1 -> AX_E_PERMIT_INVERT_1.PERMIT
- AX_SPLIT_4.OUT2 -> AX_R_TRIG_Alarm.QI

Ereignisverbindungen:

- AX_R_TRIG_Alarm.EO -> AX_SR.S
- ACK.IND -> AX_E_PERMIT_INVERT_1.EI3
- DigitalInput_CLK_I1.IND -> AX_E_PERMIT_INVERT_1.EI1
- DigitalInput_CLK_I2.IND -> AX_E_PERMIT_INVERT_1.EI2
- AX_E_PERMIT_INVERT_1.EO3 -> AX_SR.R
- AX_E_PERMIT_INVERT_1.EO3 -> Select_4_AUI.EI3
- AX_E_PERMIT_INVERT_1.EO2 -> Select_4_AUI.EI2
- AX_E_PERMIT_INVERT_1.EO1 -> Select_4_AUI.EI1

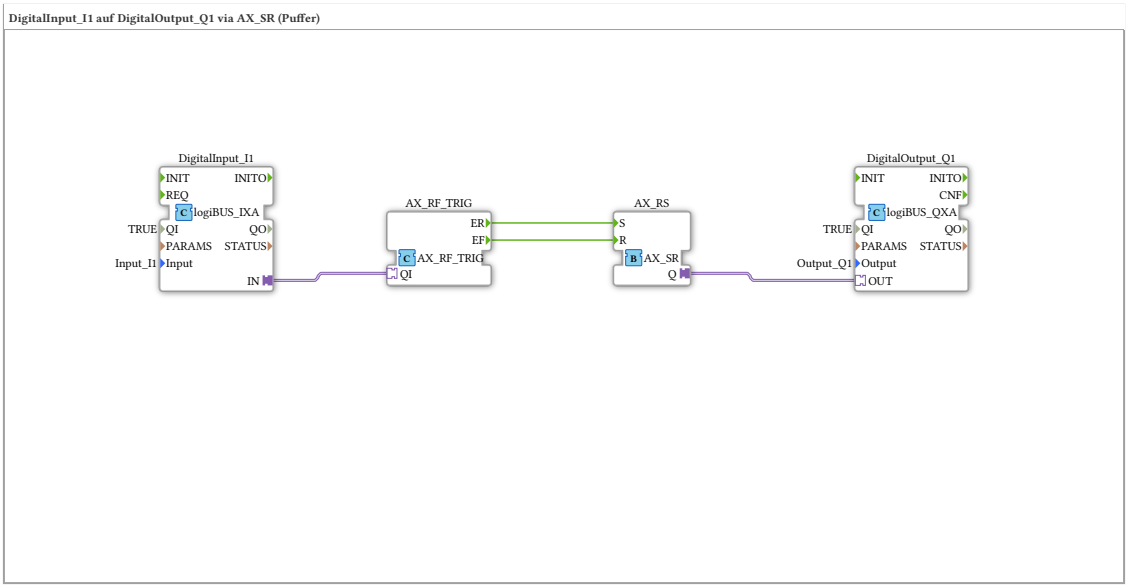
Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_019c_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.234 Uebung_020a_AX: DigitalInput_I1 auf DigitalOutput_Q1 via AX_SR (Puffer)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung leitet einen digitalen Eingang über einen Set/Reset-Speicher (`AX_SR`) auf einen digitalen Ausgang weiter. Im Gegensatz zu Uebung_001_AX, die eine direkte Verbindung nutzt, werden die Flanken des Eingangs hier explizit in Ereignisse gewandelt: die steigende Flanke setzt den Speicher, die fallende Flanke setzt ihn zurück. Im Ergebnis verhält sich die Schaltung wie eine direkte Verbindung, nutzt aber intern eine ereignisbasierte Speicherlogik.

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_I1**: logiBUS Digitaleingang
  - **Typ**: `logiBUS::io::DI::logiBUS_IXA`
  - **Parameter**: `QI = TRUE`, `Input = Input_I1`
  - **Erklärung**: Liest den physischen Eingang `Input_I1` und stellt ihn als AX-Adapterausgang `IN` bereit.
- **AX_RF_TRIG**: Flankenerkennung
  - **Typ**: `adapter::events::unidirectional::AX_RF_TRIG`
  - **Parameter**: keine
  - **Erklärung**: Wertet das Signal am Adapter `QI` aus und meldet eine steigende Flanke (0⟶1) als Ereignis `ER`, eine fallende Flanke (1⟶0) als Ereignis `EF`.
- **AX_RS**: Set/Reset-Speicher
  - **Typ**: `adapter::events::unidirectional::AX_SR`
  - **Parameter**: keine
  - **Erklärung**: Das Ereignis `S` setzt den Ausgang `Q` auf TRUE, `R` setzt ihn auf FALSE. Der Speicher hält den Zustand zwischen den Ereignissen.
- **DigitalOutput_Q1**: logiBUS Digitalausgang
  - **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
  - **Parameter**: `QI = TRUE`, `Output = Output_Q1`
  - **Erklärung**: Setzt den physischen Ausgang `Output_Q1` auf den Zustand von `AX_RS.Q`.

Programmablauf und Verbindungen

1. `DigitalInput_I1.IN` ⟶ `AX_RF_TRIG.QI` (AdapterConnection): Das Eingangssignal wird der Flankenerkennung zugeführt.
2. `AX_RF_TRIG.ER` ⟶ `AX_RS.S` (EventConnection): Die steigende Flanke setzt den Speicher.
3. `AX_RF_TRIG.EF` ⟶ `AX_RS.R` (EventConnection): Die fallende Flanke setzt den Speicher zurück.
4. `AX_RS.Q` ⟶ `DigitalOutput_Q1.OUT` (AdapterConnection): Der Speicherzustand wird auf den physischen Ausgang `Q1` gelegt.

Verhalten: Beim Drücken von `I1` setzt die steigende Flanke den Speicher, `Q1` geht an. Beim Loslassen setzt die fallende Flanke ihn zurück, `Q1` geht aus.

Anwendungsbeispiel

Dieses Muster ist die Basis für **Signalfilterung oder Entprellung**. Wenn man zwischen die Flankenerkennung und den Speicher weitere Logik (z.B. Timer) einfügt, kann man sehr präzise steuern, unter welchen Bedingungen ein Signal „einrasten“ oder „abfallen“ soll.

Zusammenfassung

Die Übung zeigt, wie ein Eingangssignal über Flankenerkennung (`AX_RF_TRIG`) und einen Set/Reset-Speicher (`AX_SR`) als Puffer auf einen Ausgang gelegt wird. Das Ergebnis entspricht einer direkten Durchschaltung, die Signalverarbeitung läuft aber ereignisbasiert.

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

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

2.1.2.235 Uebung_020b_AX: DigitalInput_I1 auf DigitalOutput_Q1; TON Einschaltverzögert; aufgelöst



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_020b_AX`. Hier wird eine Einschaltverzögerung (TON) nicht als fertiger Baustein verwendet, sondern aus diskreten Ereignis- und Speicherbausteinen aufgebaut.

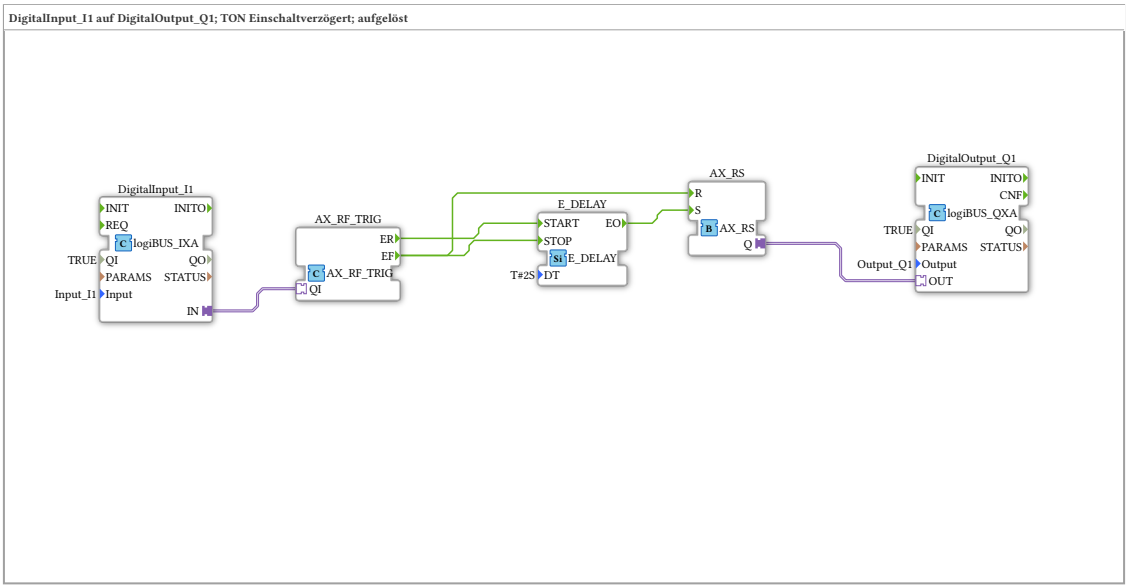
Ziel der Übung

Das Ziel dieser Übung ist es, das Verständnis für die zeitliche Steuerung von Ereignissen zu vertiefen. Anstatt den fertigen `AX_TON` Baustein (siehe `Uebung_020c_AX`) zu nutzen, wird hier demonstriert, wie ein `E_DELAY` Baustein in eine Logikschleife eingebunden werden kann, um eine identische Funktionalität zu erreichen.

Beschreibung und Komponenten

Die Subapplikation `Uebung_020b_AX.SUB` kombiniert eine Ereignis-Weiche, eine Zeitverzögerung und einen RS-Speicher.

Funktionsbausteine (FBs)



- **`DigitalInput_I1`**: Typ `logiBUS_IXA`. Signaleingang.
- **`AX_SWITCH`**: Leitet das Ereignis bei steigender Flanke an `EO1` und bei fallender Flanke an `EO0` weiter.
- **`E_DELAY`**: Verzögert ein am `START`-Eingang eintreffendes Ereignis um die Zeit `DT` hier 2 Sekunden.
- **`AX_RS`**: Der Ergebnisspeicher.
- **`DigitalOutput_Q1`**: Typ `logiBUS_QXA`. Signalausgang.

Funktionsweise

Die Logik arbeitet in drei Phasen:

1. **Einschalten (Start der Verzögerung)**: Wird `I1` gedrückt, sendet die Weiche ein Event an `E_DELAY.START`. Die Zeit läuft.
2. **Ablauf der Zeit (Durchschalten)**: Nach 2 Sekunden feuert `E_DELAY` an seinem Ausgang `EO`. Dieses Event setzt den Speicher `AX_RS.S` -> `Q1` geht an.
3. **Ausschalten (Sofort-Stopp)**: Wird `I1` losgelassen, sendet die Weiche ein Event an `EO0`. Dieses Event stoppt sofort eine eventuell laufende Zeitmessung (`E_DELAY.STOP`) und setzt gleichzeitig den Speicher zurück (`AX_RS.R`) -> `Q1` geht sofort aus.

Im Ergebnis leuchtet die Lampe nur, wenn der Taster mindestens 2 Sekunden lang gehalten wird. Wird er vorher losgelassen, passiert nichts.

Anwendungsbeispiel

Schutz gegen Fehlbedienung: Ein Taster zum Öffnen eines schweren Tors oder zum Starten einer Maschine muss 2 Sekunden lang gehalten werden, um eine versehentliche Betätigung auszuschließen.

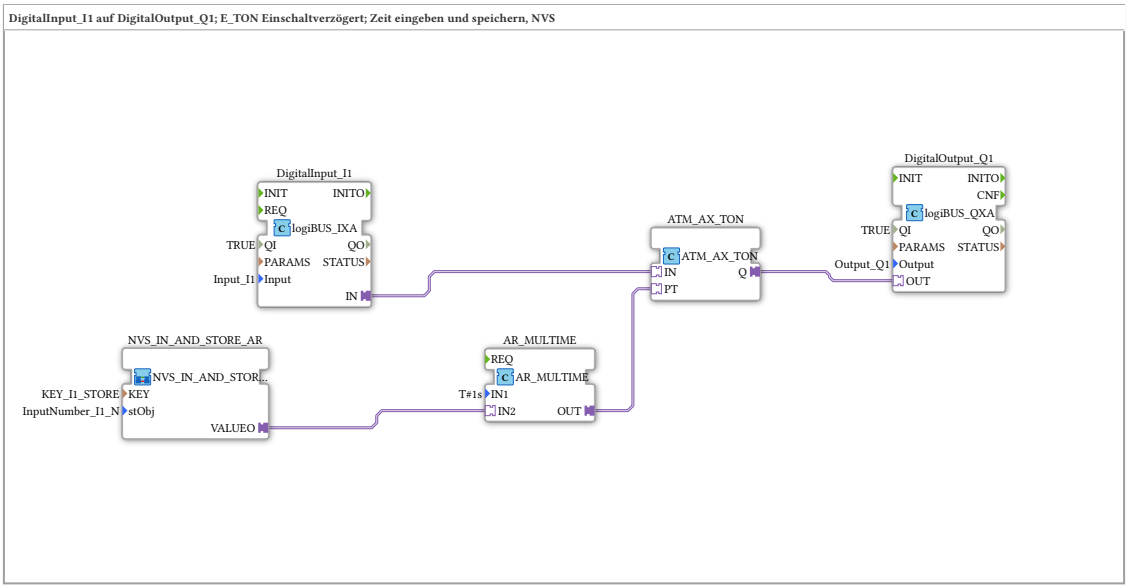
2.1.2.236 Uebung_020c2_AX: Einschaltverzögerung mit einstellbarer, NVS-gespeicherter Zeit (AX)

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_020c2_AX`. Wir bauen eine einschaltverzögerte Schaltung, deren Verzögerungszeit über das ISOBUS-Terminal eingestellt und dauerhaft im ESP32-Flash gespeichert wird.

Ziel der Übung

`DigitalInput_I1` schaltet `DigitalOutput_Q1` erst nach einer einstellbaren Verzögerung (`E_TON`, hier als AX-Adapter `ATM_AX_TON`). Die Verzögerungszeit ist über das VT-Terminal einstellbar und wird per NVS (Non-Volatile Storage, ESP32-Flash) dauerhaft gespeichert, sodass sie einen Neustart übersteht.

Beschreibung und Komponenten



- **`DigitalInput_I1`**: `logiBUS::io::DI::logiBUS_IXA`, physischer Taster/Schalter.
- **`ATM_AX_TON`**: `adapter::events::unidirectional::timers::ATM_AX_TON`, AX-Adaptervariante des Einschaltverzögerungs-Timers `E_TON`.
- **`AR_MULTIME`**: `adapter::iec61131::arithmetic::AR_MULTIME`, multipliziert den vom Bediener eingegebenen Zeitwert mit einer Zeitbasis, um die tatsächliche `PT` (Preset Time) für den Timer zu berechnen.
- **`NVS_IN_AND_STORE_AR`**: liest den vom Bediener über das VT eingegebenen Zahlenwert und speichert ihn dauerhaft im ESP32-Flash (`KEY_I1_STORE`).
- **`DigitalOutput_Q1`**: `logiBUS::io::DQ::logiBUS_QXA`.

Funktionsweise

1. Der Bediener gibt am ISOBUS-Terminal einen Zahlenwert ein, der über `NVS_IN_AND_STORE_AR` sofort im ESP32-Flash gespeichert wird.
2. `AR_MULTIME` berechnet daraus die tatsächliche Verzögerungszeit (`PT`) für `ATM_AX_TON`.
3. Sobald `DigitalInput_I1` aktiv wird, startet `ATM_AX_TON` die Verzögerung.
4. Nach Ablauf der (gespeicherten) Zeit schaltet `DigitalOutput_Q1` ein.
5. Die gespeicherte Zeit bleibt auch nach einem Neustart des Controllers erhalten.

2.1.2.237 Uebung_020c3_AX: DigitalInput_I1 auf DigitalOutput_Q1; AX_FB_TON; Einschaltverzögert

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_020c3_AX`. Hier wird der adapterbasierte IEC 61131-3 Timer-Baustein `AX_FB_TON` verwendet, der eine regelmäßige Triggerung (Takt) benötigt, um den internen Status (ET) zu aktualisieren.

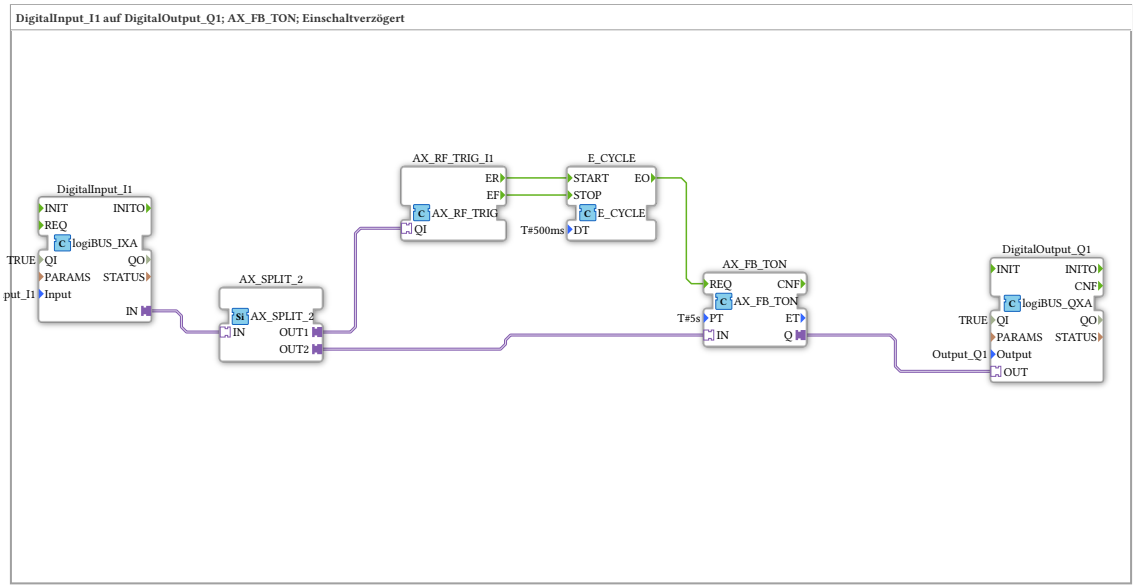
Ziel der Übung

Das Ziel ist es, eine Einschaltverzögerung mit einem klassischen SPS-Verhalten (inkl. ET-Ausgang) in einer ereignisbasierten Umgebung zu realisieren. Da `AX_FB_TON` ein zyklisches Verhalten für die Zeitberechnung erwartet, wird ein Taktgeber (`E_CYCLE`) eingesetzt.

Beschreibung und Komponenten

Die Subapplikation `Uebung_020c3_AX.SUB` nutzt einen `E_CYCLE` Baustein, um den Takt für den Timer zu generieren.

Funktionsbausteine (FBs)



- **`DigitalInput_I1`**: Liest den Eingangszustand über einen AX-Adapter ein.
- **`AX_FB_TON`**: Der Einschaltverzögerungs-Timer mit Adapter-Schnittstellen. Er benötigt zyklische Ereignisse am `REQ`-Eingang.
- **`E_CYCLE`**: Erzeugt alle 500ms ein Ereignis, solange der Eingang `I1` aktiv ist.
- **`AX_SWITCH`**: Startet und stoppt den `E_CYCLE` basierend auf dem Eingangszustand.
- **`DigitalOutput_Q1`**: Gibt das verzögerte Signal über einen AX-Adapter aus.

Funktionsweise

1. **Start**: Sobald der Taster `I1` gedrückt wird, schaltet der `AX_SWITCH` den `E_CYCLE` ein.
2. **Taktung**: Der `E_CYCLE` sendet alle 500ms ein Event an `AX_FB_TON.REQ`.
3. **Verzögerung**: Nach Ablauf von 5 Sekunden (PT) wird der Ausgang `Q` des Timers aktiv.
4. **Stopp**: Wird der Taster losgelassen, stoppt der `E_CYCLE` und der Timer wird zurückgesetzt.

Fazit

Dieses Beispiel verdeutlicht, dass Bausteine mit IEC 61131-3 Verhalten (wie die `AX_FB_*` Serie) eine kontinuierliche Ereignisquelle benötigen, um Zeitwerte wie `ET` korrekt zu verarbeiten.

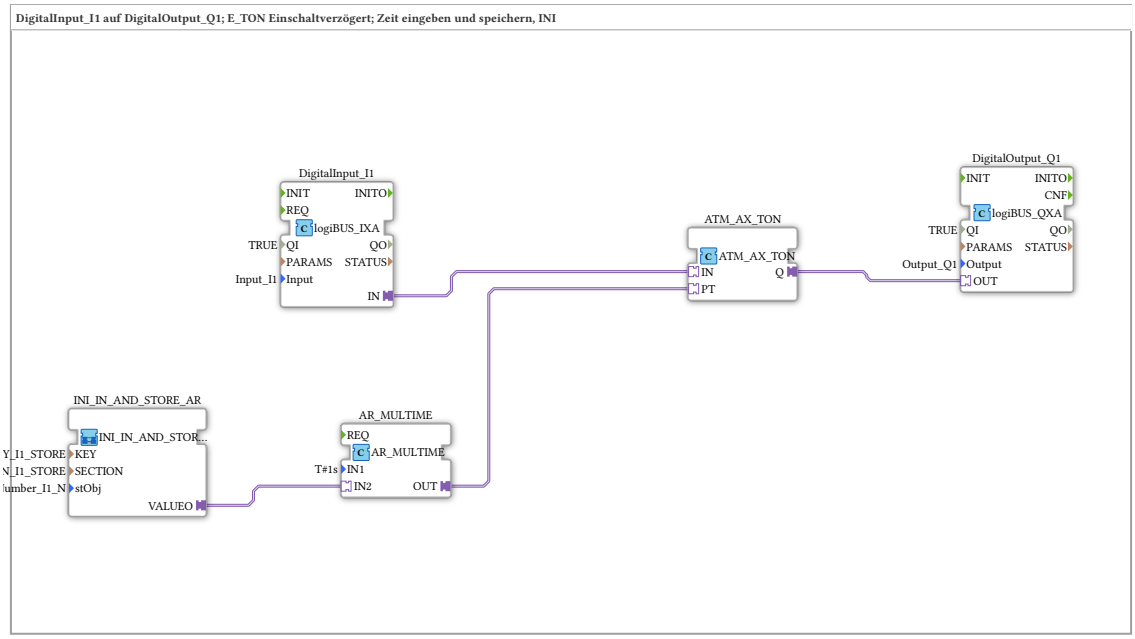
2.1.2.238 Uebung_020c4_AX: Einschaltverzögerung mit einstellbarer, INI-gespeicherter Zeit (AX)

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_020c4_AX`. Diese Übung entspricht `Uebung_020c2_AX` (siehe dort für die generelle Funktionsweise), speichert die vom Bediener eingegebene Verzögerungszeit aber statt in NVS (ESP32-Flash) in einer INI-Datei.

Ziel der Übung

`DigitalInput_I1` schaltet `DigitalOutput_Q1` erst nach einer einstellbaren, über das VT-Terminal eingegebenen Verzögerung. Die Zeit wird über `INI_IN_AND_STORE_AR` in einer INI-Datei (Abschnitt `SECTION_I1_STORE`, Schlüssel `KEY_I1_STORE`) dauerhaft gespeichert.

Beschreibung und Komponenten



- **`DigitalInput_I1`**: `logiBUS::io::DI::logiBUS_IXA`.
- **`ATM_AX_TON`**: `adapter::events::unidirectional::timers::ATM_AX_TON`, AX-Adapter-Einschaltverzögerung.
- **`AR_MULTIME`**: `adapter::iec61131::arithmetic::AR_MULTIME`, berechnet die tatsächliche `PT` (Preset Time) aus dem gespeicherten Wert.
- **`INI_IN_AND_STORE_AR`**: liest den vom Bediener eingegebenen Wert und speichert ihn in einer INI-Datei statt in NVS.
- **`DigitalOutput_Q1`**: `logiBUS::io::DQ::logiBUS_QXA`.

Funktionsweise

1. Der Bediener gibt am ISOBUS-Terminal einen Zahlenwert ein, der über `INI_IN_AND_STORE_AR` sofort in die INI-Datei geschrieben wird.
2. `AR_MULTIME` berechnet daraus die tatsächliche Verzögerungszeit (`PT`) für `ATM_AX_TON`.
3. Sobald `DigitalInput_I1` aktiv wird, startet `ATM_AX_TON` die Verzögerung.
4. Nach Ablauf der gespeicherten Zeit schaltet `DigitalOutput_Q1` ein.
5. Die gespeicherte Zeit übersteht einen Neustart des Controllers.

2.1.2.239 Uebung_020c_AX: DigitalInput_I1 auf DigitalOutput_Q1; AX_TON; Einschaltverzögert



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_020c_AX`.

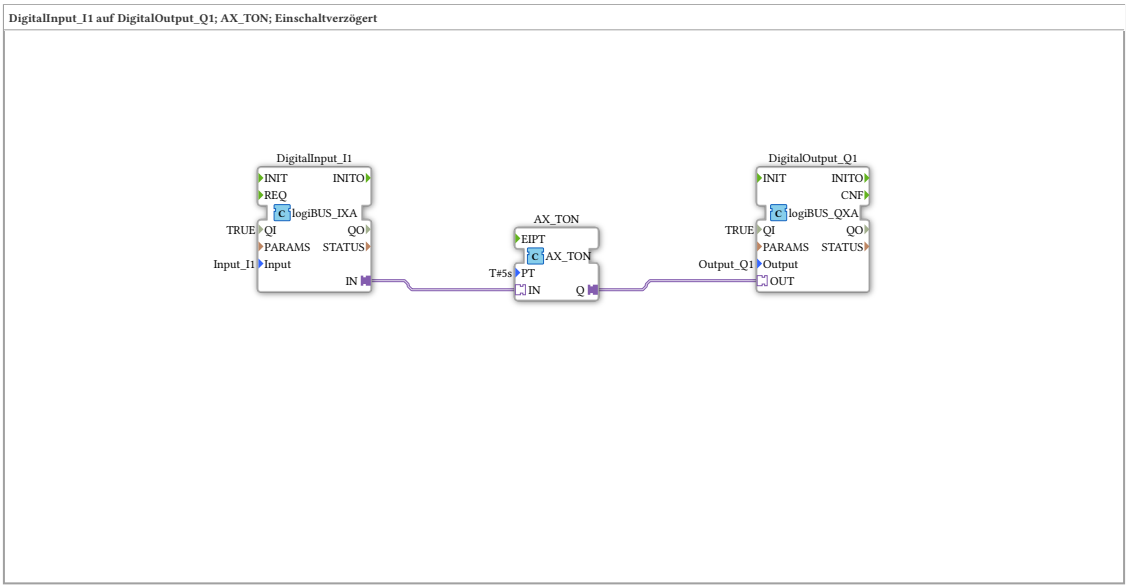
Ziel der Übung

Kennenlernen des Timer-Bausteins `AX_TON`.

Beschreibung und Komponenten

Die Subapplikation `Uebung_020c_AX.SUB` verzögert das Einschaltsignal.

Funktionsbausteine (FBs)



- **`AX_TON`**: Timer On-Delay.
- **Parameter `PT`**: Preset Time (hier 5 Sekunden).

Funktionsweise

1. Eingang `I1` wird TRUE.
2. `AX_TON` startet die Zeitmessung.
3. Nach 5 Sekunden wird der Ausgang `Q` TRUE -> Lampe geht an.
4. Wird `I1` vor Ablauf der 5s wieder FALSE, bricht der Timer ab und die Lampe bleibt aus.
5. Beim Ausschalten von `I1` geht die Lampe sofort aus (keine Ausschaltverzögerung).

Anwendungsbeispiel

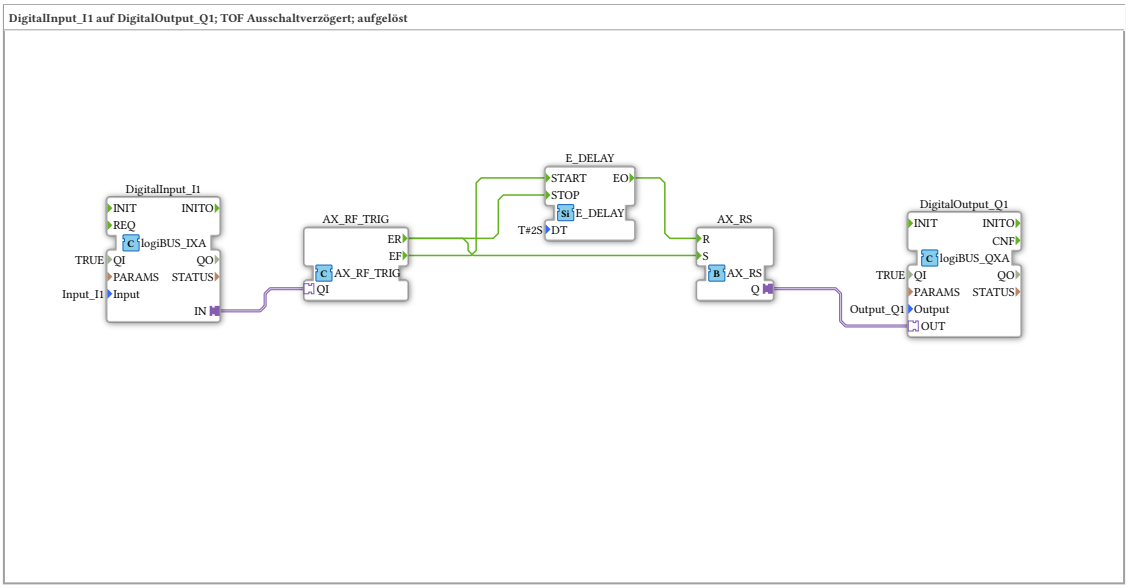
Anlaufwarnung: Bevor ein Förderband startet, ertönt für 5 Sekunden eine Hupe. Erst danach läuft der Motor an.

2.1.2.240 Uebung_020d_AX: DigitalInput_I1 auf DigitalOutput_Q1; TOF Ausschaltverzögert; aufgelöst

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_020d_AX`. Hier wird eine Ausschaltverzögerung (TOF) aus diskreten Ereignis- und Speicherbausteinen aufgebaut.

Ziel der Übung



Das Ziel dieser Übung ist die Analyse der Ausschaltverzögerung auf Logikebene. Im Gegensatz zur Einschaltverzögerung (`Uebung_020b_AX`) startet hier die Zeitmessung erst beim *Loslassen* des Tasters.

Beschreibung und Komponenten

Die Subapplikation `Uebung_020d_AX.SUB` nutzt eine Ereignis-Weiche, um den Speicher beim Drücken sofort zu setzen und beim Loslassen zeitverzögert zurückzusetzen.

Funktionsbausteine (FBs)

- **`DigitalInput_I1`**: Typ `logiBUS_IXA`. Signaleingang.
- **`AX_SWITCH`**: Trennt steigende (`EO1`) und fallende (`EO0`) Flanken.
- **`AX_RS`**: Der Ergebnisspeicher.
- **`E_DELAY`**: Verzögert das Rücksetz-Ereignis um 2 Sekunden (`DT = T#2S`).
- **`DigitalOutput_Q1`**: Typ `logiBUS_QXA`. Signalausgang.

Funktionsweise

Die Logik arbeitet wie folgt:

1. **Einschalten (Sofort)**: Wird `I1` gedrückt, sendet die Weiche ein Event an `EO1`. Dieses Event setzt sofort den Speicher `AX_RS.S` -> `Q1` geht an. Gleichzeitig wird eine eventuell noch laufende Verzögerung gestoppt (`E_DELAY.STOP`).
2. **Loslassen (Start der Verzögerung)**: Wird `I1` losgelassen, sendet die Weiche ein Event an `EO0`. Dieses Event startet die Zeitmessung `E_DELAY.START`. Der Speicher bleibt vorerst auf TRUE.
3. **Ausschalten (Nach Ablauf der Zeit)**: Nach 2 Sekunden feuert `E_DELAY.EO`. Dieses Event setzt den Speicher zurück (`AX_RS.R`) -> `Q1` geht aus.

Im Ergebnis leuchtet die Lampe sofort beim Drücken und bleibt nach dem Loslassen noch genau 2 Sekunden lang an.

Anwendungsbeispiel

Nachlaufsteuerung: Ein Treppenhauslicht oder ein Lüfter soll sofort anspringen, wenn der Schalter betätigt wird, aber nach dem Verlassen des Raumes noch für eine gewisse Zeit weiterlaufen.

2.1.2.241 Uebung_020e2_AX: DigitalInput_I1 auf DigitalOutput_Q1; AX_FB_TOF; Ausschaltverzögert

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_020e2_AX`. Hier wird der adapterbasierte IEC 61131-3 Timer-Baustein `AX_FB_TOF` verwendet, der eine regelmäßige Triggerung (Takt) benötigt.

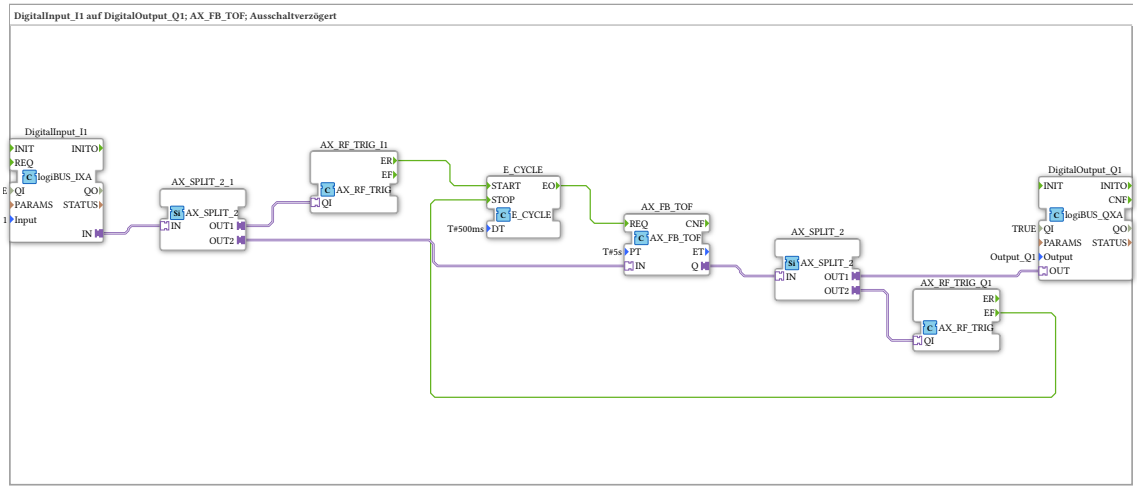
Ziel der Übung

Realisierung einer Ausschaltverzögerung, die auch während der Nachlaufzeit ihren Status (ET) aktualisiert.

Beschreibung und Komponenten

Die Subapplikation `Uebung_020e2_AX.SUB` nutzt einen `E_CYCLE` Baustein für die Taktung.

Funktionsbausteine (FBs)



- **`AX_FB_TOF`**: Der Ausschaltverzögerungs-Timer.
- **`E_CYCLE`**: Liefert den Takt (500ms) für den Timer.
- **`AX_SWITCH_I1`**: Startet den Takt bei Aktivierung des Eingangs.
- **`AX_SWITCH_Q1`**: Stoppt den Takt erst dann, wenn auch der Ausgang des Timers wieder abgefallen ist (Nachlauf beendet).

Funktionsweise

1. **Aktivierung**: Bei `I1 = TRUE` wird der Ausgang sofort aktiv und der Taktgeber startet.
2. **Nachlauf**: Fällt `I1` ab, läuft der Timer weiter. Der `E_CYCLE` bleibt aktiv, da der Ausgang `Q` noch `TRUE` ist.
3. **Abschluss**: Sobald die 5 Sekunden abgelaufen sind, fällt `Q` ab und der `E_CYCLE` wird gestoppt.

Fazit

Die Übung zeigt die komplexe Ansteuerung eines Ausschaltverzögerers, bei dem der Taktgeber über die gesamte Dauer (Einschaltzeit + Nachlaufzeit) aktiv bleiben muss.

2.1.2.242 Uebung_020e_AX: DigitalInput_I1 auf DigitalOutput_Q1; AX_TOF; Ausschaltverzögert

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_020e_AX`.

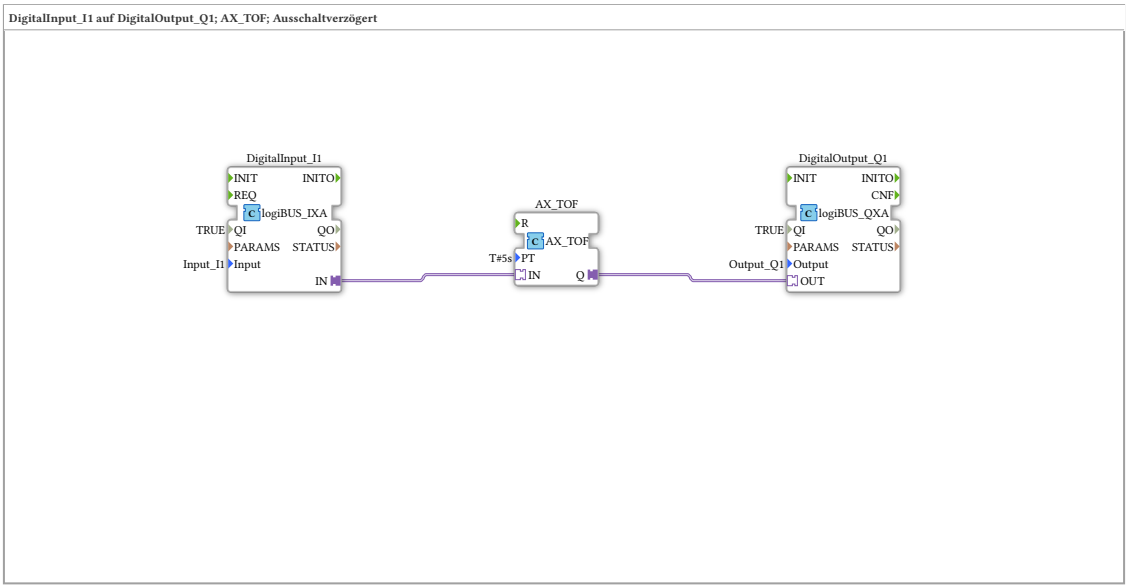
Ziel der Übung

Kennenlernen des Timer-Bausteins `AX_TOF`.

Beschreibung und Komponenten

Die Subapplikation `Uebung_020e_AX.SUB` verzögert das Ausschaltsignal.

Funktionsbausteine (FBs)



- **`AX_TOF`**: Timer Off-Delay.
- **Parameter `PT`**: Preset Time (hier 5 Sekunden).

Funktionsweise

1. Eingang `I1` wird TRUE -> Lampe geht **sofort** an.
2. Eingang `I1` wird FALSE -> Timer startet.
3. Nach 5 Sekunden wird der Ausgang `Q` FALSE -> Lampe geht aus.

Anwendungsbeispiel

Nachlauf: Ein Lüfter im Bad läuft noch 5 Minuten nach, nachdem das Licht ausgeschaltet wurde.

2.1.2.243 Uebung_020f2_AX: DigitalInput_I1 auf DigitalOutput_Q1; AX_FB_TP; Impuls-formend

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_020f2_AX`. Hier wird der adapterbasierte IEC 61131-3 Timer-Baustein `AX_FB_TP` verwendet, der eine regelmäßige Triggerung (Takt) benötigt.

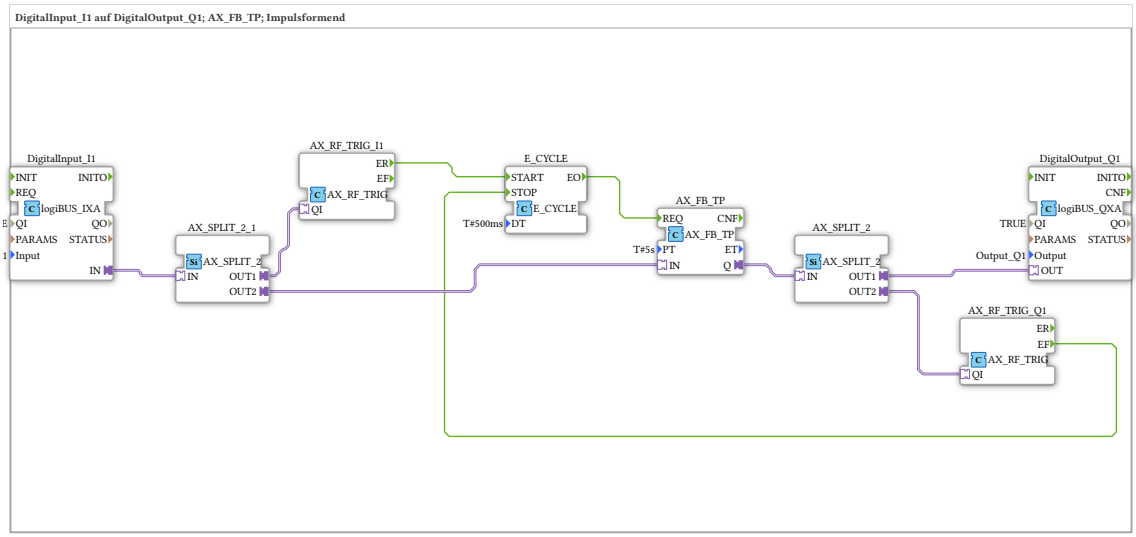
Ziel der Übung

Das Ziel ist es, die Brücke zwischen der klassischen SPS-Programmierung (zyklisch) und der IEC 61499 (ereignisbasiert) zu schlagen. Da ein `AX_FB_TP` intern die Zeit zählt, muss sein `REQ`-Eingang regelmäßig mit Ereignissen versorgt werden, solange der Timer läuft.

Beschreibung und Komponenten

Die Subapplikation `Uebung_020f2_AX.SUB` nutzt einen `E_CYCLE` Baustein, um den Takt für den Timer zu generieren.

Funktionsbausteine (FBs)



- **`AX_FB_TP`**: Der Impuls-Timer mit Adapter-Schnittstellen. Er reagiert auf die steigende Flanke am Eingang und hält den Ausgang für die Zeit `PT` auf TRUE.
- **`E_CYCLE`**: Erzeugt alle 500ms ein Ereignis, um den Timer zu aktualisieren.
- **`AX_SWITCH`**: Überwacht den Status, um den Taktgeber bei Bedarf zu starten oder zu stoppen.

Funktionsweise

1. **Start**: Beim Drücken des Tasters wird der Impuls am `AX_FB_TP` ausgelöst und der `E_CYCLE` gestartet.
2. **Taktung**: Der `E_CYCLE` sendet alle 500ms ein Event an `AX_FB_TP.REQ`. Bei jedem Event prüft der Timer, wie viel Zeit vergangen ist und aktualisiert seinen Status.
3. **Impuls**: Solange der Impuls läuft, bleibt der Ausgang `Q` wahr.
4. **Stopp**: Sobald die 5 Sekunden abgelaufen sind, stoppt der `E_CYCLE` über eine entsprechende Logik (`AX_SWITCH`).

Dieses Beispiel zeigt, dass klassische Bausteine zwar verwendet werden können, aber einen höheren Aufwand für die Ereignisverwaltung erfordern als spezialisierte Adapter-Bausteine (wie `AX_TP`).

Anwendungsbeispiel

Integration von Legacy-Code: Wenn bereits fertige Bausteine aus der „alten“ SPS-Welt übernommen werden sollen, die auf zyklische Abarbeitung angewiesen sind, ist dieses Takt-Muster unerlässlich.

2.1.2.244 Uebung_020f3_AX: DigitalInput_I1 auf DigitalOutput_Q1; BLINKER

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_020f3_AX`.

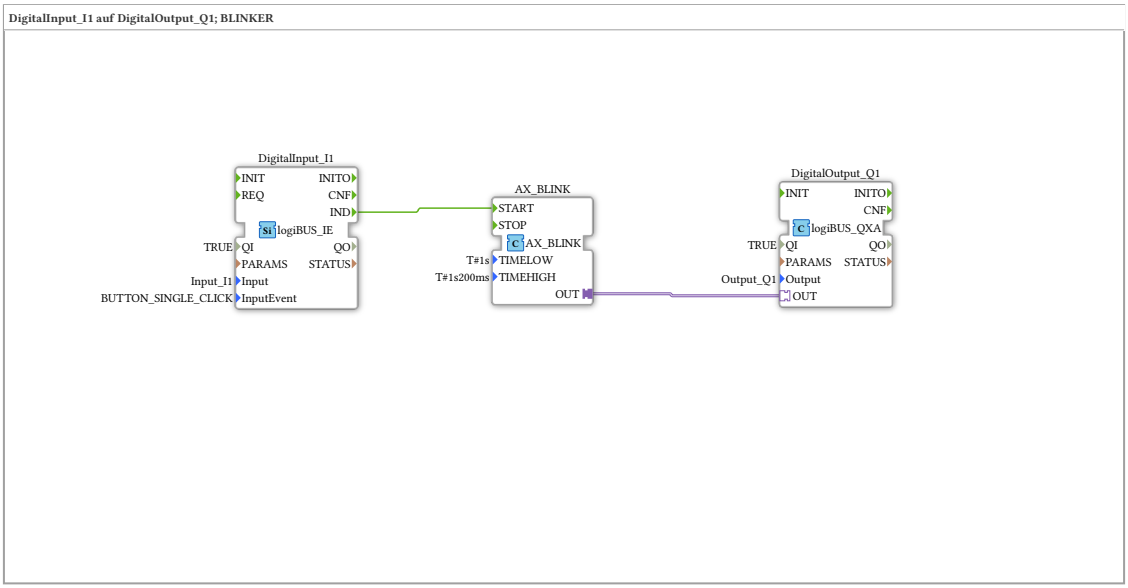
Ziel der Übung

Verwendung des `AX_BLINK` Bausteins für asymmetrisches Blinken.

Beschreibung und Komponenten

Die Subapplikation `Uebung_020f3_AX.SUB` nutzt einen spezialisierten Blinker-Baustein.

Funktionsbausteine (FBs)



- **`AX_BLINK`**: Erzeugt ein Blinksignal.
- **Parameter `TIMELOW`**: Zeit für „Aus“ (1s).
- **Parameter `TIMEHIGH`**: Zeit für „An“ (1.2s).

Funktionsweise

Ein Event am Eingang `START` (hier vom Taster `I1` geliefert) startet den Blinker. Er läuft dann mit den parametrierten Zeiten. Der Baustein integriert im Grunde die Logik von zwei Timern und einem Flip-Flop.

Anwendungsbeispiel

Fehlercode: Eine LED blinkt in einem bestimmten Muster (kurz an, lang aus), um einen Fehlercode zu signalisieren.

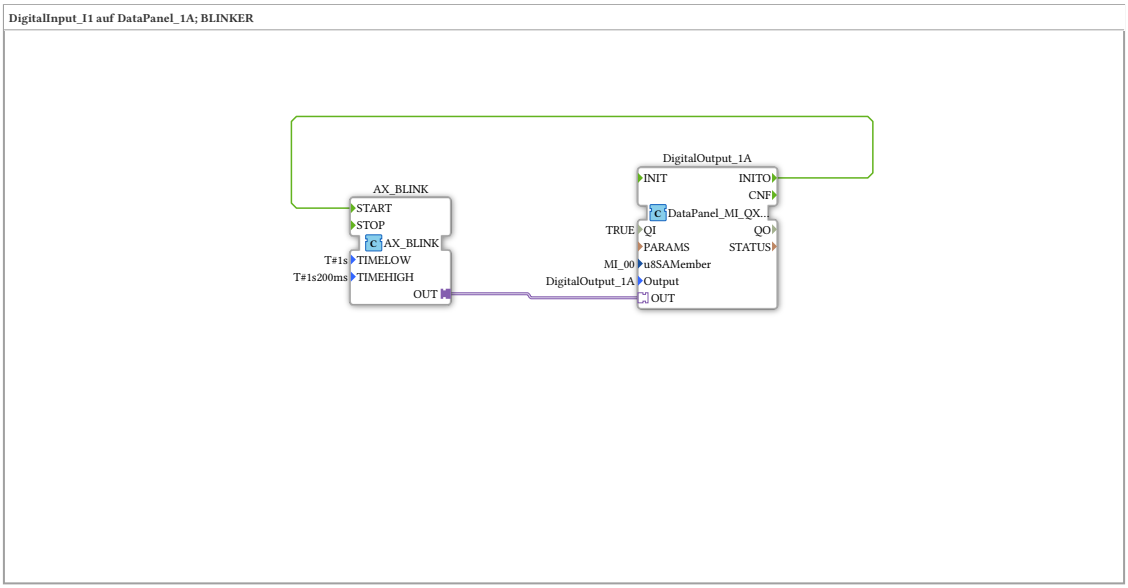
2.1.2.245 Uebung_020f4_AX: Blinklicht auf DataPanel-Ausgang (AX)

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_020f4_AX`. Wir bauen einen einfachen Blinker auf einem DataPanel-Ausgang.

Ziel der Übung

Ein DataPanel-Ausgang (`DigitalOutput_1A`) blinkt dauerhaft mit fest eingestellten Ein-/Aus-Zeiten, sobald er initialisiert wurde.

Beschreibung und Komponenten



- **`DigitalOutput_1A`**: `DataPanel::io::MI::DQ::DataPanel_MI_QXA`, Ausgang auf dem DataPanel-Erweiterungsmodul (`u8SAMember = MI_00`).
- **`AX_BLINK`**: `adapter::events::unidirectional::signals::AX_BLINK`, erzeugt ein periodisches Ein/Aus-Signal mit `TIMELOW = T#1s` und `TIMEHIGH = T#1s200ms`.

Funktionsweise

1. Nach der Initialisierung des Ausgangs (`DigitalOutput_1A.INITO`) startet `AX_BLINK`.
2. `AX_BLINK` schaltet den Ausgang abwechselnd für `T#1s200ms` ein und für `T#1s` aus (Blinken).
3. Der AX-Adapter überträgt den Blinkzustand direkt an `DigitalOutput_1A.OUT`.

2.1.2.246 Uebung_020f_AX: DigitalInput_I1 auf DigitalOutput_Q1; AX_TP; Impulsformend

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_020f_AX`. Hier wird ein Impuls-Timer (TP - Timer Pulse) verwendet, um eine definierte Einschaltdauer zu erzwingen.

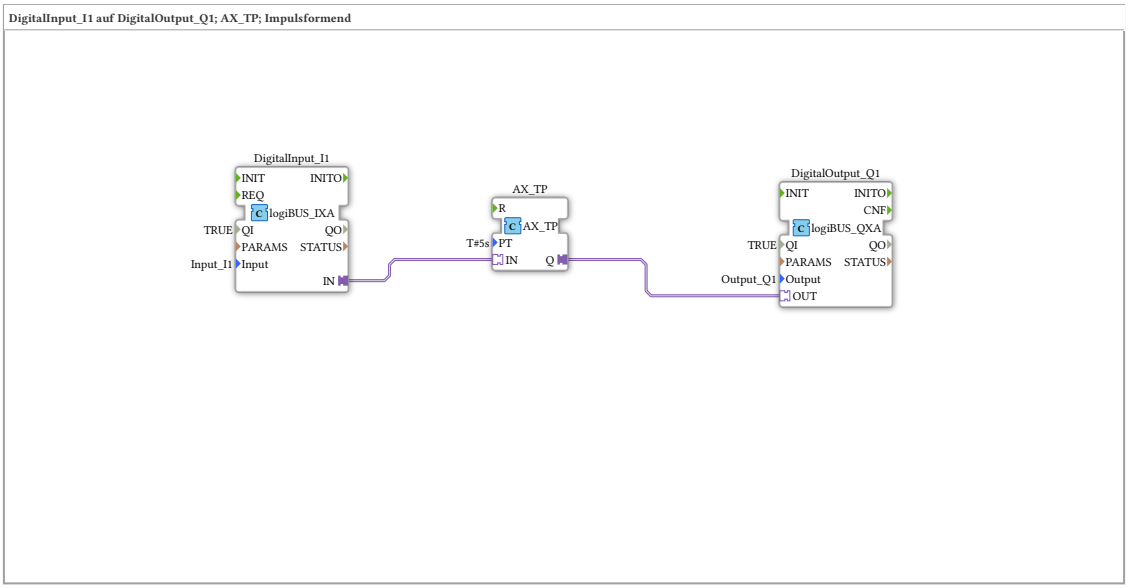
Ziel der Übung

Das Ziel dieser Übung ist die Anwendung des `AX_TP` Bausteins. Ein Impulsglied ist ideal, wenn eine Aktion für eine exakte Zeitspanne ausgeführt werden soll, unabhängig davon, wie lange der auslösende Taster gedrückt bleibt.

Beschreibung und Komponenten

Die Subapplikation `Uebung_020f_AX.SUB` nutzt einen Adapter-Timer vom Typ `AX_TP`.

Funktionsbausteine (FBs)



- **`DigitalInput_I1`**: Typ `logiBUS_IXA`. Der Auslöser.
- **`AX_TP`**: Erzeugt bei einer steigenden Flanke am Eingang einen Impuls der Länge `PT` (hier 5 Sekunden) am Ausgang `Q`.
- **`DigitalOutput_Q1`**: Typ `logiBUS_QXA`. Der Aktor.

Funktionsweise

1. **Triggerung**: Sobald der Eingang `I1` den Zustand `TRUE` einnimmt, startet der Timer.
2. **Aktivierung**: Der Ausgang `Q` wird sofort `TRUE` und die Lampe `Q1` leuchtet.
3. **Zeitablauf**: Auch wenn der Nutzer den Taster sofort wieder loslässt (oder ihn 10 Sekunden lang gedrückt hält), bleibt die Lampe für exakt **5 Sekunden** an.
4. **Abschluss**: Nach Ablauf der 5 Sekunden geht die Lampe automatisch aus. Ein neuer Impuls kann erst nach dem nächsten Flankenwechsel am Eingang ausgelöst werden.

Anwendungsbeispiel

Schmierung oder Reinigung: Eine Zentralschmierung an einer Maschine oder eine Reinigungsdüse soll nach einem Startsignal für genau 5 Sekunden aktiv sein, um die korrekte Menge an Medium abzugeben.

2.1.2.247 Uebung_020g_AX: DigitalInput_I1 auf DigitalOutput_Q1; AX_TONOF; Ein-/Ausschaltverzögert

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_020g_AX`. Hier wird der kombinierte Verzögerungsbaustein `AX_TONOF` verwendet.

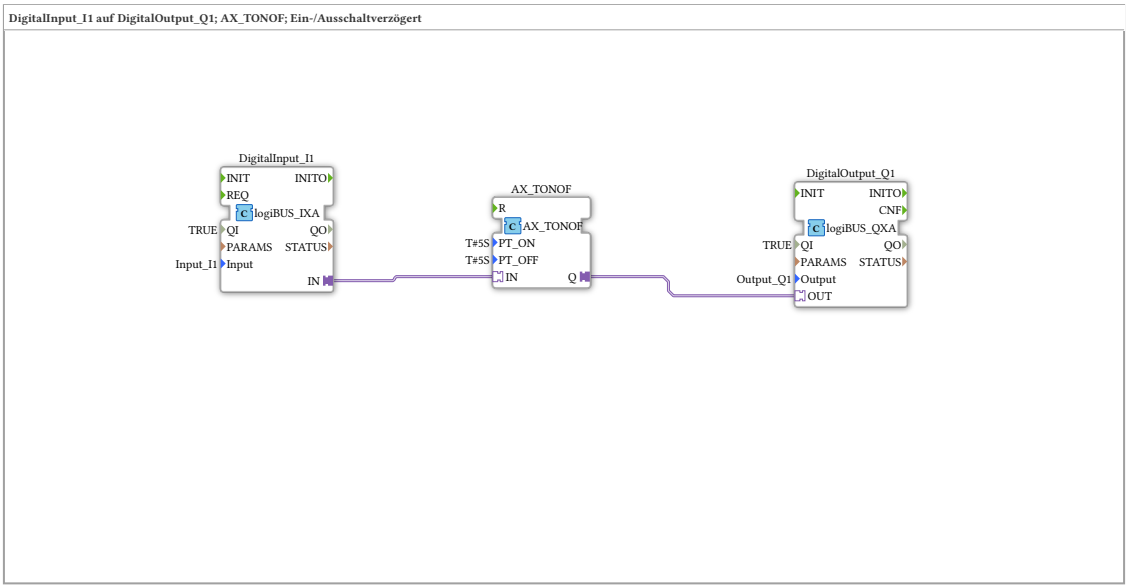
Ziel der Übung

Das Ziel ist es, ein Signal in beide Richtungen zeitlich zu filtern. Kurze Impulse (sowohl positive als auch negative) werden ignoriert. Nur wenn ein Zustand länger als die definierte Zeit stabil anliegt, wird er an den Ausgang weitergegeben.

Beschreibung und Komponenten

Die Subapplikation `Uebung_020g_AX.SUB` nutzt den Baustein `AX_TONOF`.

Funktionsbausteine (FBs)



- **`DigitalInput_I1`**: Typ `logiBUS_IXA`.
- **`AX_TONOF`**: Vereint Einschaltverzögerung (`PT_ON`) und Ausschaltverzögerung (`PT_OFF`) in einem Baustein. Hier sind beide Zeiten auf 5 Sekunden eingestellt.
- **`DigitalOutput_Q1`**: Typ `logiBUS_QXA`.

Funktionsweise

1. **Einschalten**: Wird `I1` gedrückt, passiert am Ausgang zunächst nichts. Erst nach **5 Sekunden** dauerhaften Drückens schaltet `Q1` ein.
2. **Ausschalten**: Wird `I1` losgelassen, bleibt `Q1` zunächst an. Erst nach **5 Sekunden** im losgelassenen Zustand schaltet `Q1` aus.

Kurzes Antippen (< 5s) führt nicht zum Einschalten. Kurzes Loslassen (< 5s) führt nicht zum Ausschalten.

Anwendungsbeispiel

Füllstandsüberwachung: Ein Schwimmerschalter in einem Tank, in dem das Medium schwappt. Die Pumpe soll erst einschalten, wenn der Sensor 5 Sekunden lang „Leer“ meldet, und erst ausschalten, wenn er 5 Sekunden lang „Voll“ meldet. Dies verhindert ein nervöses Flattern der Pumpe bei Wellenbewegungen.

2.1.2.248 Uebung_020h_AX: DigitalInput_I1 auf DigitalOutput_Q1; AX_PULSE; Impulsformend



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_020h_AX`. Hier wird der Baustein `AX_PULSE` verwendet, der im Gegensatz zum `AX_TP` rein ereignisbasiert arbeitet.

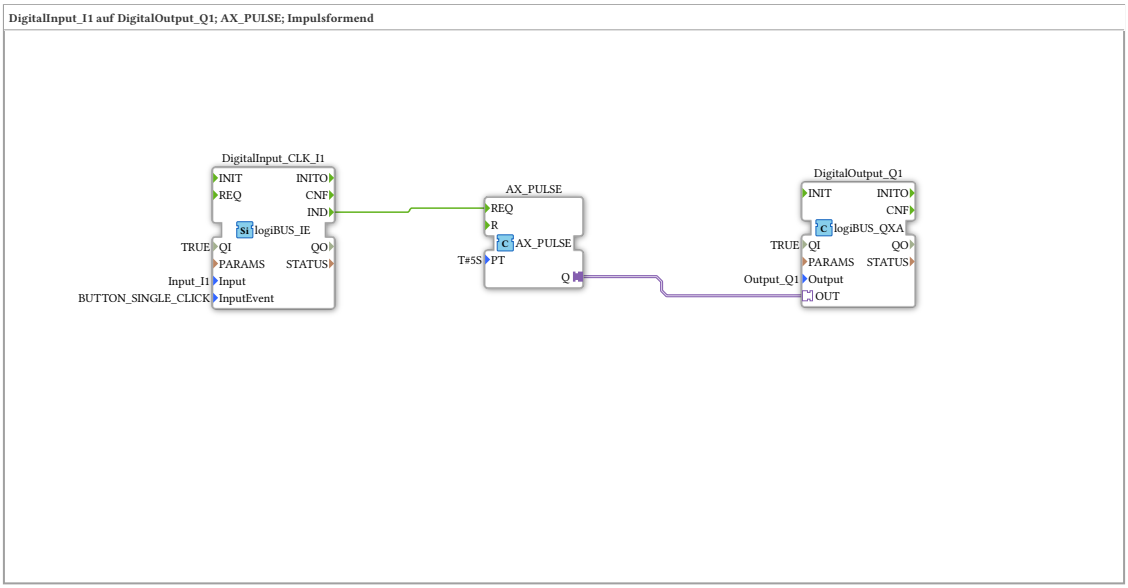
Ziel der Übung

Das Ziel ist es, ein einzelnes, kurzes Ereignis (z.B. einen Mausklick oder Taster-Impuls) in eine länger anhaltende Aktion zu verwandeln. Der Fokus liegt hierbei auf der rein ereignisorientierten Schnittstelle des Bausteins.

Beschreibung und Komponenten

Die Subapplikation `Uebung_020h_AX.SUB` kombiniert einen Event-Eingang mit einem Adapter-Puls-Baustein.

Funktionsbausteine (FBs)



- **`DigitalInput_CLK_I1`**: Typ `logiBUS_IE`. Liefert ein Ereignis bei einem Einfachklick (`BUTTON_SINGLE_CLICK`).
- **`AX_PULSE`**: Startet einen Timer bei Eintreffen eines Ereignisses am `REQ`-Eingang. Der Ausgang `Q` bleibt für die Zeit `PT` (5 Sekunden) auf TRUE.
- **`DigitalOutput_Q1`**: Typ `logiBUS_QXA`.

Funktionsweise

1. **Ereignis**: Der Nutzer klickt kurz auf den Taster `I1`.
2. **Trigger**: Der Eingangsbaustein sendet ein `IND`-Event an den `REQ`-Eingang von `AX_PULSE`.
3. **Aktion**: Der Timer startet sofort. Der Adapter-Ausgang `Q` wird `TRUE` und schaltet die Lampe `Q1` ein.
4. **Autarkie**: Da der Baustein ereignisbasiert ist, muss der Eingang nicht gehalten werden. Er „merkt“ sich den Startimpuls.
5. **Ende**: Nach 5 Sekunden schaltet der Ausgang automatisch wieder auf `FALSE`.

Anwendungsbeispiel

Türöffner: Ein kurzer Tastendruck an der Gegensprechanlage löst einen elektrischen Türöffner für genau 5 Sekunden aus, damit der Besucher eintreten kann.

2.1.2.249 Uebung_020i_AX: DigitalInput_I1 auf DigitalOutput_Q1; AX_PULSE; Teach-In



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_020i_AX`. Diese Übung kombiniert Zeitmessung und Zeitsteuerung zu einer lernfähigen Puls-Funktion.

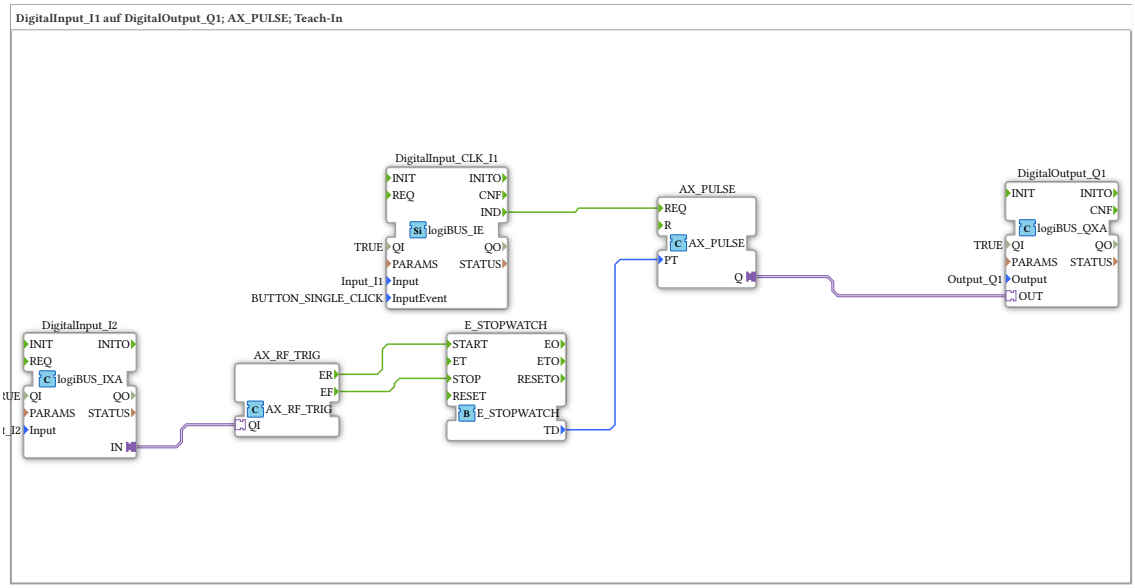
Ziel der Übung

Das Ziel ist die Implementierung eines Teach-In-Verfahrens. Anstatt die Zeit `PT` fest im Programm zu hinterlegen, kann der Bediener die gewünschte Dauer durch Halten eines Tasters vorgeben. Die Steuerung speichert diese Zeit und wendet sie bei zukünftigen Impulsen an.

Beschreibung und Komponenten

Die Subapplikation `Uebung_020i_AX.SUB` nutzt eine Stoppuhr, um die Zeitvorgabe für einen Puls-Baustein dynamisch zu ändern.

Funktionsbausteine (FBs)



- **`DigitalInput_I2` (Teach-Taste)**: Typ `logiBUS_IXA`. Misst, wie lange der Taster gedrückt wird.
- **`AX_SWITCH`**: Wandelt das Drücken/Loslassen von `I2` in Start/Stopp-Signale für die Stoppuhr um.
- **`E_STOPWATCH`**: Misst die Zeit zwischen `START` und `STOP` und gibt die Dauer am Ausgang `TD` aus.
- **`AX_PULSE`**: Der Impuls-Baustein. Sein Zeitparameter `PT` ist mit dem gemessenen Wert `TD` der Stoppuhr verbunden.
- **`DigitalInput_CLK_I1` (Start-Taste)**: Typ `logiBUS_IE`. Löst den Puls aus.
- **`DigitalOutput_Q1`**: Typ `logiBUS_QXA`. Signalausgang.

Funktionsweise

Die Anwendung erfolgt in zwei Schritten:

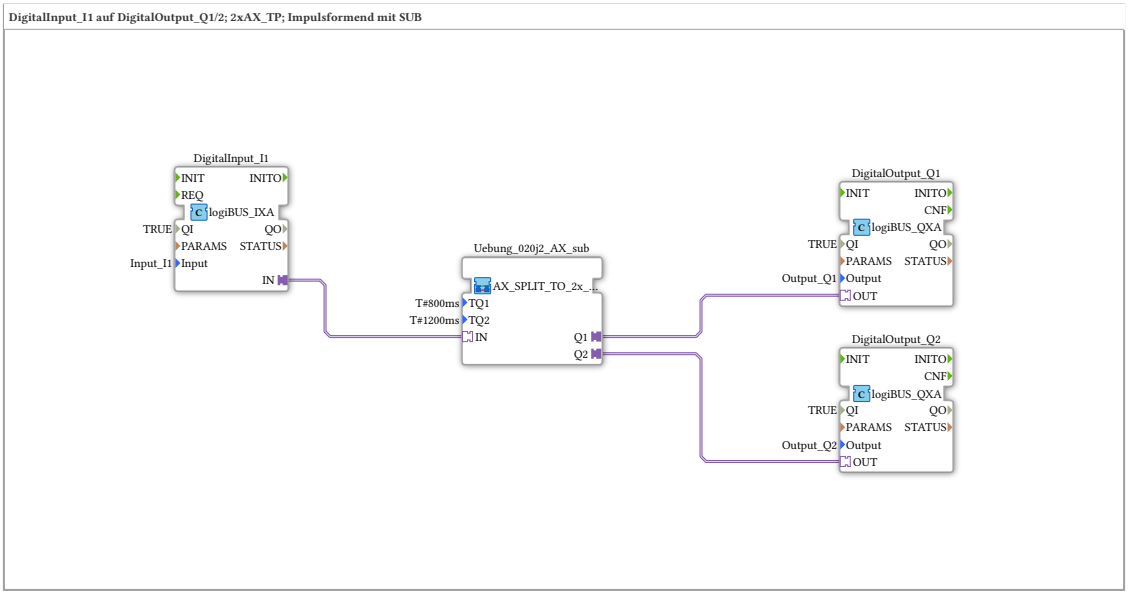
1. **Anlernen (Teach-In)**: Der Nutzer hält Taster `I2` für die gewünschte Dauer gedrückt (z.B. 3,5 Sekunden).
   - Beim Drücken startet `E_STOPWATCH`.
   - Beim Loslassen stoppt die Messung. Der Wert (3,5s) liegt nun am Eingang `PT` von `AX_PULSE` an.
2. **Ausführen**: Der Nutzer klickt kurz auf Taster `I1`.
   - `AX_PULSE` wird getriggert und schaltet die Lampe `Q1` für exakt die zuvor gelernten 3,5 Sekunden ein.

Jedes neue Anlernen über `I2` überschreibt die gespeicherte Zeit für den nächsten Puls.

Anwendungsbeispiel

Dosiersteuerung: Ein Landwirt möchte die Ausbringmenge eines Zusatzstoffs manuell kalibrieren. Er hält den Befüllknopf so lange, bis die gewünschte Testmenge erreicht ist. Die Steuerung merkt sich diese Zeit und kann den Dosiervorgang ab nun per einfachem Tastendruck exakt wiederholen.

2.1.2.250 Uebung_020j2_AX: DigitalInput_I1 auf DigitalOutput_Q1/2; 2xAX_TP; Impulsformend mit SUB



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert die Verarbeitung von Signalen mittels Adapter-Verbindungen (AX) innerhalb einer IEC 61499 Anwendung. Ein digitales Eingangssignal (`DigitalInput_I1`) wird verwendet, um zwei separate digitale Ausgänge (`DigitalOutput_Q1` und `DigitalOutput_Q2`) anzusteuern. Die Besonderheit dieser Übung liegt in der Verwendung einer gekapselten Sub-Applikation (`Uebung_020j2_AX_sub`), die das Eingangssignal aufteilt und zwei unabhängige Impulsgeber (Pulse Timer) ansteuert.

Verwendete Funktionsbausteine (FBs)

Im Hauptnetzwerk werden folgende Bausteine verwendet:

- **DigitalInput_I1** (`logiBUS::io::DI::logiBUS_IXA`): Repräsentiert den physischen Eingang `Input_I1`.
- **DigitalOutput_Q1** (`logiBUS::io::DQ::logiBUS_QXA`): Repräsentiert den physischen Ausgang `Output_Q1`.
- **DigitalOutput_Q2** (`logiBUS::io::DQ::logiBUS_QXA`): Repräsentiert den physischen Ausgang `Output_Q2`.
- **Uebung_020j2_AX_sub** (`Uebungen::Uebung_020j2_AX_sub`): Eine benutzerdefinierte Sub-Applikation, welche die Logik zur Impulsformung und Signalverteilung enthält.

Sub-Bausteine: Uebung_020j2_AX_sub

Diese Sub-Applikation kapselt die Logik für das Splitten des Adapter-Signals und das Erzeugen der Zeitimpulse.

- **Typ**: SubAppType
- **Schnittstellen**:
  - **Eingangsvariablen**: `TQ1` (Zeitwert für Timer 1), `TQ2` (Zeitwert für Timer 2).
  - **Adapter-Eingang (Socket)**: `IN` (Typ: `AX`).
  - **Adapter-Ausgänge (Plugs)**: `Q1`, `Q2` (Typ: `AX`).
- **Verwendete interne FBs**:
  - **AX_SPLIT_2**: `adapter::events::unidirectional::AX_SPLIT_2`
    - **Funktion**: Teilt eine eingehende Adapter-Verbindung auf zwei ausgehende Adapter-Verbindungen auf (OUT1 und OUT2).
    - **Verbindung**: Der Eingang `IN` der Sub-App ist mit dem Eingang dieses Bausteins verbunden.
  - **AX_TP_Q1**: `adapter::events::unidirectional::timers::AX_TP`

    - **Parameter**: PT (Pulse Time) wird durch den Sub-App-Eingang TQ1 gesetzt.
    - **Eingang**: Verbunden mit OUT1 des AX_SPLIT_2.
    - **Ausgang**: Verbunden mit dem Adapter-Plug Q1 der Sub-App.
  - **AX_TP_Q2**: adapter::events::unidirectional::timers::AX_TP
    - **Parameter**: PT (Pulse Time) wird durch den Sub-App-Eingang TQ2 gesetzt.
    - **Eingang**: Verbunden mit OUT2 des AX_SPLIT_2.
    - **Ausgang**: Verbunden mit dem Adapter-Plug Q2 der Sub-App.
- **Funktionsweise**: Das eingehende Signal am Adapter IN wird durch den AX_SPLIT_2 Baustein vervielfältigt. Beide Signalpfade durchlaufen anschließend jeweils einen eigenen Impuls-Timer (AX_TP). Diese Timer erzeugen einen Impuls definierter Länge (TQ1 bzw. TQ2) auf der Adapter-Leitung, sobald ein Eingangssignal erkannt wird. Die resultierenden Signale werden an die Ausgänge Q1 und Q2 geleitet.

Programmablauf und Verbindungen

Der Ablauf der Übung gestaltet sich wie folgt:

1. **Signaleingang**: Das System liest den Zustand des digitalen Eingangs Input_I1 über den Baustein DigitalInput_I1.
2. **Verarbeitung in der Sub-Applikation**:
   - Die Adapter-Verbindung des Eingangs wird an die Sub-Applikation Uebung_020j2_AX_sub weitergeleitet.
   - Innerhalb der Sub-Applikation wird das Signal aufgeteilt.
   - Es werden zwei unabhängige Timer gestartet, deren Zeitwerte über Parameter im Hauptnetzwerk definiert sind:
     - **Pfad 1**: Zeitdauer T#800ms (Parameter TQ1).
     - **Pfad 2**: Zeitdauer T#1200ms (Parameter TQ2).
3. **Signalausgabe**:
   - Der Ausgang Q1 der Sub-Applikation (800ms Impuls) steuert den DigitalOutput_Q1.
   - Der Ausgang Q2 der Sub-Applikation (1200ms Impuls) steuert den DigitalOutput_Q2.

Dadurch wird erreicht, dass ein einzelnes Eingangssignal zwei Ausgänge aktiviert, die jedoch unterschiedlich lange aktiv bleiben (Impulsformung).

Zusammenfassung

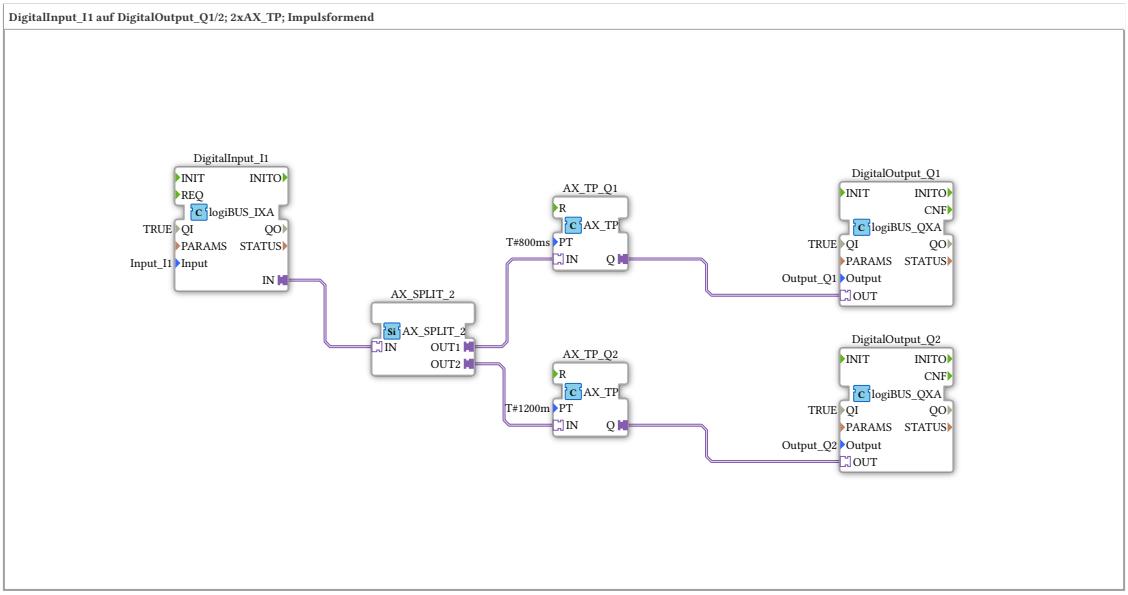
Die Übung Uebung_020j2_AX zeigt anschaulich die Verwendung von unidirektionalen Adapter-Verbindungen (AX) zur Signalverarbeitung. Sie demonstriert, wie Logik in Sub-Applikationen gekapselt werden kann, um komplexe Funktionen (hier: Signal-Splitting und parallele Timer-Steuerung) übersichtlich und wiederverwendbar zu gestalten. Lernziele sind der Umgang mit dem AX_SPLIT Baustein sowie die Parametrierung von Timer-Bausteinen (AX_TP) über Sub-Applikations-Schnittstellen.

2.1.2.251 Uebung_020j2_AX_sub

Hinweis

Diese Übung existiert nicht mehr als eigene Übung. Der Baustein wurde in die Bibliothek verschoben und heißt jetzt AX_SPLIT_TO_2x_AX_TP. Beschreibung, Schnittstelle und Netzwerk stehen dort.

2.1.2.252 Uebung_020j_AX: DigitalInput_I1 auf DigitalOutput_Q1/2; 2xAX_TP; Impulsformend



• – ○ • – ○ • – ○ •

Einleitung

Die Übung **Uebung_020j_AX** demonstriert die Verwendung von Adapter-Verbindungen zur Signalverarbeitung. Ein digitales Eingangssignal (`Input_I1`) wird eingelesen, aufgeteilt und zur Ansteuerung von zwei digitalen Ausgängen (`Output_Q1` und `Output_Q2`) verwendet. Dabei kommen impulsformende Zeitglieder (Timer) zum Einsatz, die über Adapter-Schnittstellen kommunizieren.

Verwendete Funktionsbausteine (FBs)

In dieser Übung werden verschiedene Bausteine innerhalb des Netzwerks verschaltet, um die gewünschte Logik zu realisieren.

Sub-Bausteine: DigitalInput_I1

- **Typ**: `logiBUS::io::DI::logiBUS_IXA`
- **Beschreibung**: Dieser Baustein stellt die Schnittstelle zum physischen Eingang her.
- **Parameter**:
  - `QI` = `TRUE` (Baustein aktiviert)
  - `Input` = `Input_I1` (Zuordnung der Hardware-Ressource)

Sub-Bausteine: AX_SPLIT_2

- **Typ**: `adapter::events::unidirectional::AX_SPLIT_2`
- **Beschreibung**: Ein Splitter-Baustein für Adapter-Verbindungen. Er nimmt eine eingehende Adapter-Verbindung entgegen und teilt sie auf zwei Ausgänge auf, um das Signal an mehrere Empfänger weiterzuleiten.

Sub-Bausteine: AX_TP_Q1

- **Typ**: `adapter::events::unidirectional::timers::AX_TP`
- **Beschreibung**: Ein Impuls-Timer (Pulse Timer) basierend auf Adapter-Technologie. Er erzeugt einen Impuls definierter Länge.
- **Parameter**:
  - `PT` = `T#800ms` (Impulsdauer von 800 Millisekunden)

Sub-Bausteine: AX_TP_Q2

- **Typ**: `adapter::events::unidirectional::timers::AX_TP`
- **Beschreibung**: Ein zweiter Impuls-Timer für den zweiten Ausgangspfad.
- **Parameter**:

  - `PT` = `T#1200m` (Impulsdauer von 1200 Minuten – *Hinweis: Hierbei handelt es sich laut IEC 61131-3 Syntax um Minuten. Im Kontext von Übungen ist oft Millisekunden (ms) gemeint, der Code spezifiziert jedoch* `m`).

Sub-Bausteine: DigitalOutput_Q1

- **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
- **Beschreibung**: Schnittstelle zum ersten physischen Ausgang.
- **Parameter**:
  - `QI` = `TRUE`
  - `Output` = `Output_Q1`

Sub-Bausteine: DigitalOutput_Q2

- **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
- **Beschreibung**: Schnittstelle zum zweiten physischen Ausgang.
- **Parameter**:
  - `QI` = `TRUE`
  - `Output` = `Output_Q2`

Programmablauf und Verbindungen

Der Ablauf der Übung gestaltet sich wie folgt:

1. **Signaleingang**: Das Signal wird über den Baustein `DigitalInput_I1` (Ressource `Input_I1`) in das System geholt.
2. **Signalverteilung**: Die Adapter-Verbindung vom Eingang (`IN`) geht auf den Eingang des `AX_SPLIT_2` Bausteins. Dieser vervielfältigt die Adapter-Information auf zwei Ausgänge (`OUT1` und `OUT2`).
3. **Signalverarbeitung Pfad 1**:
   - Der Ausgang `OUT1` des Splitters ist mit dem Timer `AX_TP_Q1` verbunden.
   - Sobald ein Signalereignis eintritt, generiert dieser Timer einen Impuls von **800 ms**.
   - Der Ausgang des Timers (`Q`) steuert direkt den `DigitalOutput_Q1` an.
4. **Signalverarbeitung Pfad 2**:
   - Der Ausgang `OUT2` des Splitters ist mit dem Timer `AX_TP_Q2` verbunden.
   - Dieser Timer ist auf eine Dauer von **1200 m** (Minuten) konfiguriert.
   - Der Ausgang dieses Timers (`Q`) steuert den `DigitalOutput_Q2` an.

Lernziele:

- Verständnis von Adapter-Konzepten in 4diac (`AX`-Bausteine).
- Verwendung von `AX_SPLIT`-Bausteinen zur Verzweigung von Daten- und Ereignisflüssen, die in Adaptern gekapselt sind.
- Parametrierung von Adapter-Timern (`AX_TP`).

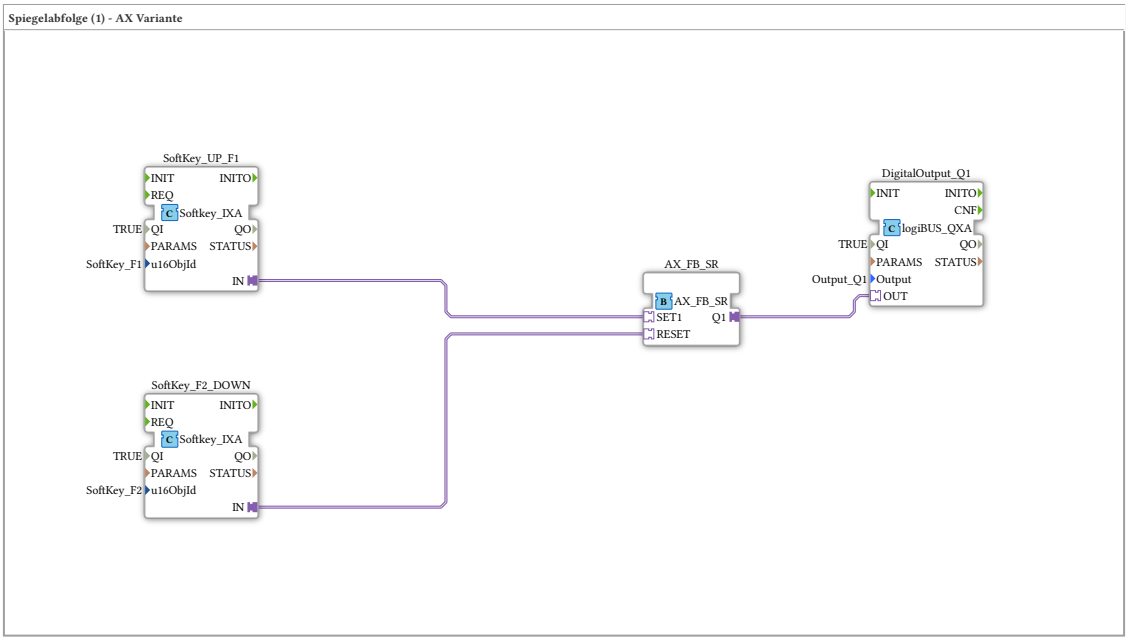
Zusammenfassung

Die Übung `Uebung_020j_AX` zeigt eine Parallelschaltung von zwei Ausgängen, die durch einen gemeinsamen Eingang ausgelöst werden. Durch die Verwendung von Adapter-Timern werden unterschiedliche Impulsdauern für `Q1` und `Q2` realisiert, ohne dass klassische Event- und Data-Connections separat gezogen werden müssen. Besonderes Augenmerk liegt auf der korrekten Verwendung des Splitter-Bausteins und der Zeit-Syntax der Parameter.

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

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de
- 🌐 Gesamtwiderstand in Reihen- & Parallelschaltung auf ms-muc-docs.de

2.1.2.253 Uebung_021_AX: Spiegelabfolge (1) - AX Variante



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung realisiert eine einfache Steuerung für eine **Spiegelabfolge (1)** – AX Variante. Mithilfe zweier Softkeys (F1 und F2) kann ein digitaler Ausgang gesetzt und zurückgesetzt werden. Der Funktionsbaustein bildet eine Art **Start/Stopp-Logik** für einen Antrieb (AX), der durch die Softkeys angesteuert wird. Die Übung vermittelt grundlegende Kenntnisse über die Nutzung von Softkeys, SR-Flipflops und digitalen Ausgängen in der 4diac-IDE.

Verwendete Funktionsbausteine (FBs)

Softkey_UP_F1

- **Typ**: `isobus::UT::io::Softkey::Softkey_IXA`
- **Parameter**:
  - `QI` = TRUE (Baustein aktiv)
  - `u16ObjId` = `SoftKey_F1` (verwendet die im Pool definierte Taste F1)
- **Adapterausgang**: `IN` – wird mit dem Set-Eingang des SR-Flipflops verbunden.
- **Funktion**: Sendet bei Betätigung der Softkey-F1 ein Signal (Impuls) an den angeschlossenen Adapter.

SoftKey_F2_DOWN

- **Typ**: `isobus::UT::io::Softkey::Softkey_IXA`
- **Parameter**:
  - `QI` = TRUE
  - `u16ObjId` = `SoftKey_F2` (Taste F2)
- **Adapterausgang**: `IN` – verbunden mit dem Reset-Eingang des SR-Flipflops.
- **Funktion**: Sendet bei Betätigung der Softkey-F2 ein Signal an das SR-Glied.

AX_FB_SR

- **Typ**: `adapter::iec61131::bistableElements::AX_FB_SR`
- **Parameter**: keine weiteren
- **Adaptereingänge**:
  - `SET1` – wird durch SoftKey_UP_F1 angesteuert
  - `RESET` – wird durch SoftKey_F2_DOWN angesteuert
- **Adapterausgang**:
  - `Q1` – Ausgangssignal, das den digitalen Ausgang ansteuert

- **Funktion**: Ein **SR-Flipflop** (setzen – rücksetzen). Solange der Set-Eingang aktiv ist, bleibt Q1 = TRUE. Ein Signal am Reset-Eingang setzt Q1 auf FALSE.

DigitalOutput_Q1

- **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
- **Parameter**:
  - `QI` = TRUE (Baustein aktiv)
  - `Output` = `Output_Q1` (die physische oder virtuelle Ausgangsadresse)
- **Adaptereingang**: `OUT` – empfängt das Steuersignal vom SR-Flipflop.
- **Funktion**: Steuert den digitalen Ausgang Q1 entsprechend des anliegenden Signals (TRUE ⟶ Ausgang aktiv, FALSE ⟶ Ausgang inaktiv).

Programmablauf und Verbindungen

1. **Initialzustand**: Der SR-Ausgang `Q1` ist FALSE, der digitale Ausgang `Output_Q1` ist inaktiv.
2. **Start (Softkey F1)**: Wird die Taste **F1** gedrückt (kommentiert als „START-Knopf“), erzeugt `SoftKey_UP_F1` einen Impuls an `AX_FB_SR.SET1`. Dadurch wird der Flipflop gesetzt: `Q1` wird TRUE und bleibt es – auch wenn die Taste losgelassen wird.
3. **Stopp (Softkey F2)**: Wird die Taste **F2** gedrückt (kommentiert als „Endlage“), sendet `SoftKey_F2_DOWN` einen Impuls an `AX_FB_SR.RESET`. Der Flipflop wird zurückgesetzt: `Q1` wird FALSE, der Ausgang schaltet ab.

Die Verbindungen sind als **Adapterverbindungen** realisiert (Pfeile zwischen den Adaptern):

- `SoftKey_UP_F1.IN` ⟶ `AX_FB_SR.SET1`
- `SoftKey_F2_DOWN.IN` ⟶ `AX_FB_SR.RESET`
- `AX_FB_SR.Q1` ⟶ `DigitalOutput_Q1.OUT`

Lernziele:

- Einrichtung und Parametrierung von Softkeys in 4diac
- Verwendung eines SR-Flipflops zur speichernden Logik
- Ansteuerung eines digitalen Ausgangs
- Verständnis von Adapterverbindungen (Kommunikation zwischen FBs)

Benötigte Vorkenntnisse: Grundlegende Bedienung der 4diac-IDE, Kenntnis der Bibliotheken `isobus` und `logiBUS`.

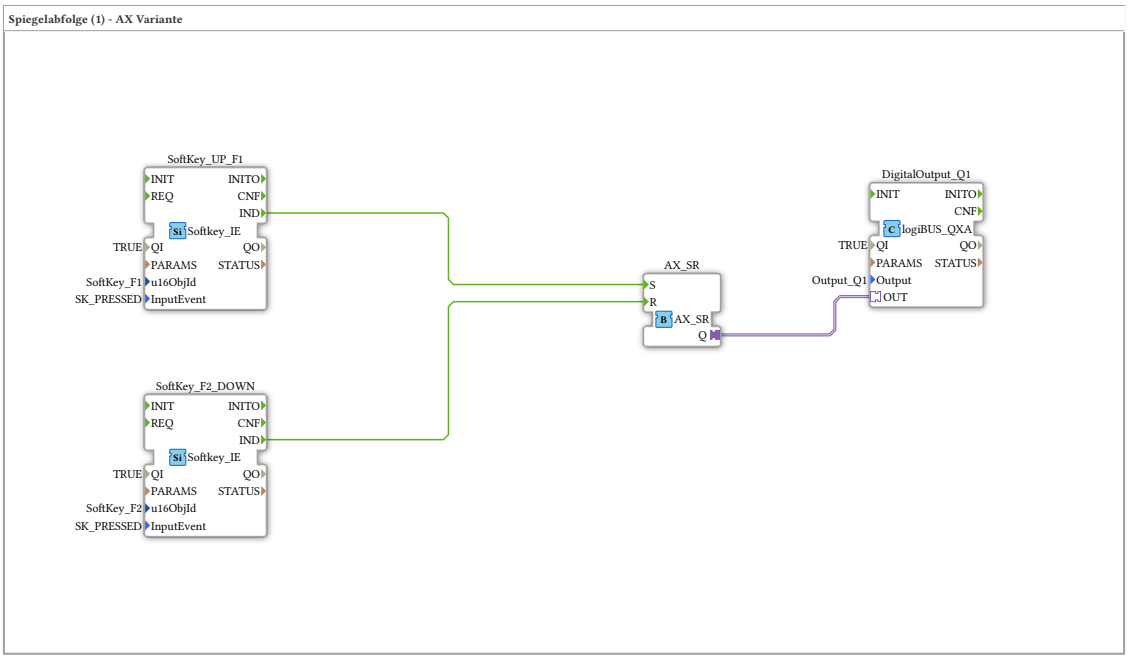
Zusammenfassung

Die Übung **Uebung_021_AX** demonstriert eine einfache Spiegelabfolge zur Steuerung eines digitalen Ausgangs über zwei Softkeys. Ein SR-Flipflop dient als speicherndes Element, das durch Taste F1 gesetzt und durch Taste F2 zurückgesetzt wird. Der Ausgang Q1 wird entsprechend geschaltet. Sie eignet sich als Einstieg in die Signalverarbeitung mit bistabilen Gliedern und in die Nutzung von Softkey-FBs zur Mensch-Maschine-Kommunikation.

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

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

2.1.2.254 Uebung_021b_AX2: Spiegelabfolge (1) - AX Variante



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung demonstriert die Steuerung einer einfachen Spiegelabfolge mithilfe eines AX-Flipflops (AX_SR). Über zwei Softkeys (F1 und F2) wird der Ablauf gestartet bzw. zurückgesetzt. Der Ausgang des Flipflops steuert einen digitalen Ausgang (Output_Q1), der z. B. einen Spiegelantrieb ansteuern kann. Die Übung zeigt den grundlegenden Umgang mit adapterbasierten Ereignis-Flipflops und digitalen Ausgängen in der 4diac-IDE.

Verwendete Funktionsbausteine (FBs)

- **DigitalOutput_Q1**
  - **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
  - **Parameter**:
    - `QI` = `TRUE`
    - `Output` = `Output_Q1`
  - **Ereigniseingänge**: `OUT` (Adapter)
  - **Funktion**: Schaltet den physikalischen Ausgang `Output_Q1` auf TRUE, sobald am Adaptereingang ein TRUE-Signal anliegt.
- **SoftKey_UP_F1**
  - **Typ**: `isobus::UT::io::Softkey::Softkey_IE`
  - **Parameter**:
    - `QI` = `TRUE`
    - `u16ObjId` = `SoftKey_F1` (Softkey F1)
    - `InputEvent` = `SK_PRESSED`
  - **Ereignisausgänge**: `IND`
  - **Funktion**: Erzeugt ein Ereignis am Ausgang `IND`, sobald die zugeordnete Softkey-Taste (F1) gedrückt wird. Der Softkey ist ständig freigegeben (QI=TRUE).
- **AX_FB_SR**
  - **Typ**: `adapter::events::unidirectional::AX_SR`
  - **Parameter**: Keine Parameter
  - **Ereigniseingänge**: `S` (Set), `R` (Reset)
  - **Adapterausgang**: `Q`

  - **Funktion**: Ein asynchroner SR-Flipflop-Adapter. Bei einem Ereignis auf `S` wird der Ausgang `Q` auf TRUE gesetzt, bei einem Ereignis auf `R` auf FALSE zurückgesetzt. Der Ausgang bleibt bis zum nächsten Ereignis erhalten.

- **SoftKey_F2_DOWN**
  - **Typ**: `isobus::UT::io::Softkey::Softkey_IE`
  - **Parameter**:
    - `QI` = `TRUE`
    - `u16ObjId` = `SoftKey_F2` (Softkey F2)
    - `InputEvent` = `SK_PRESSED`
  - **Ereignisausgänge**: `IND`
  - **Funktion**: Wie SoftKey_UP_F1, aber reagiert auf Softkey F2. Erzeugt ein Ereignis bei Tastendruck auf F2.

Programmablauf und Verbindungen

Der Ablauf ist in zwei Kommentarfelder unterteilt:

- **START-Knopf** (bei Softkey F1)
- **Endlage** (bei Softkey F2)

Ereignisverbindungen

- `SoftKey_UP_F1.IND` ⟶ `AX_FB_SR.S` Ein Druck auf Softkey F1 sendet ein Ereignis an den Set-Eingang des Flipflops.
- `SoftKey_F2_DOWN.IND` ⟶ `AX_FB_SR.R` Ein Druck auf Softkey F2 sendet ein Ereignis an den Reset-Eingang des Flipflops.

Adapterverbindung

- `AX_FB_SR.Q` ⟶ `DigitalOutput_Q1.OUT` Der Ausgang des Flipflops wird als Adaptersignal an den digitalen Ausgangsbaustein weitergeleitet. Liegt Q auf TRUE, schaltet der Ausgang `Output_Q1` durch.

Funktionsweise

1. **Start der Spiegelabfolge**: Drücken von Softkey F1 ⟶ Setzt das AX-Flipflop. Der Ausgang Q wird TRUE ⟶ der digitale Ausgang schaltet ein (z. B. Spiegel fährt aus).
2. **Endlage / Reset**: Drücken von Softkey F2 ⟶ Reset des Flipflops. Q wird FALSE ⟶ der digitale Ausgang schaltet aus (Spiegel fährt zurück).
3. Der Zustand bleibt erhalten, bis der andere Softkey gedrückt wird.

Zusammenfassung

In dieser Übung wird ein AX-SR-Flipflop verwendet, um eine einfache Spiegelsteuerung zu realisieren. Sie lernen:

- Wie man Softkey-Ereignisse in die Steuerungslogik einbindet.
- Die Funktionsweise eines adapterbasierten SR-Flipflops (Setzen und Rücksetzen über Ereignisse).
- Die Ansteuerung eines digitalen Ausgangs (logiBUS_QXA) über Adapterverbindungen.
- Grundlegende Ereignisverdrahtung und Parameterkonfiguration in der 4diac-IDE.

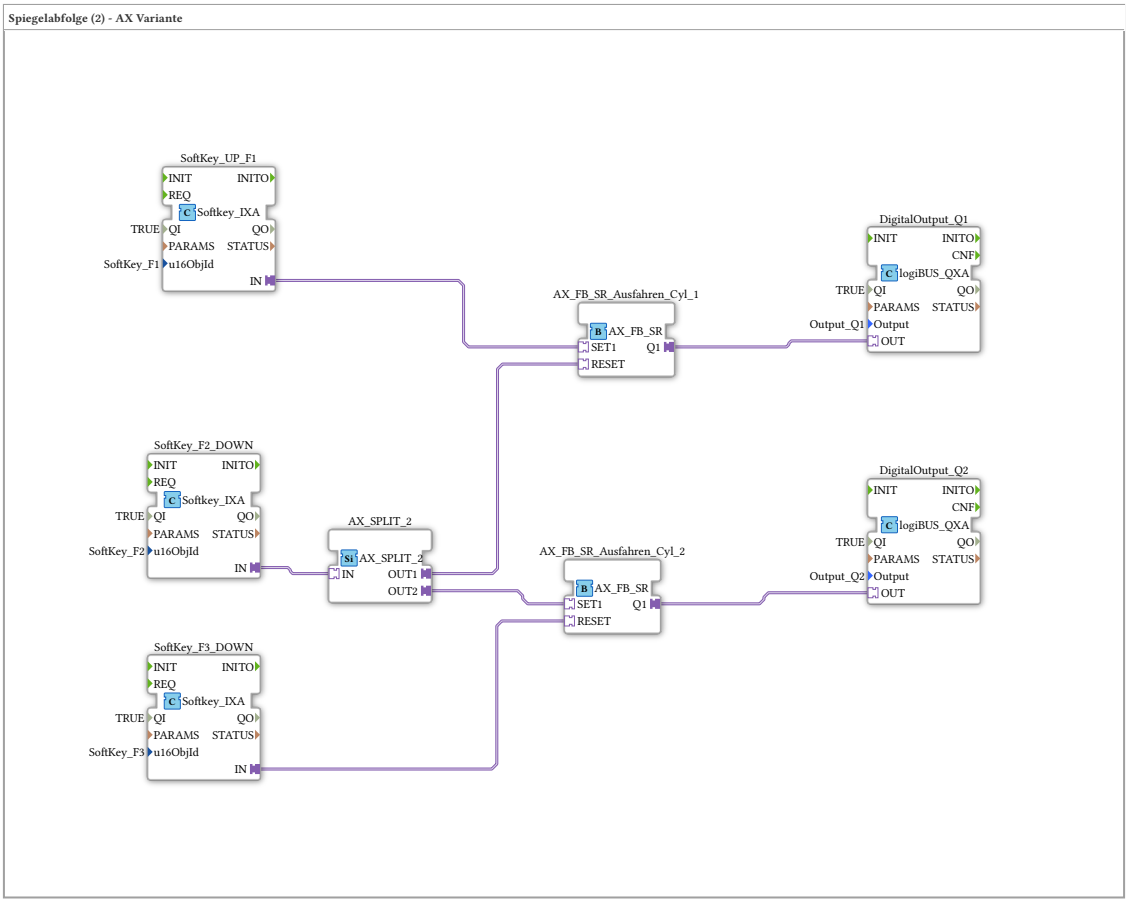
Die Übung eignet sich für Einsteiger in die ereignisgesteuerte Programmierung mit 4diac und legt das Fundament für komplexere Ablaufsteuerungen.

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

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

2.1.2.255 Uebung_022_AX: Spiegelabfolge (2) - AX Variante

Kein Bild verfügbar.



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung realisiert eine **Spiegelabfolge** für zwei Zylinder unter Verwendung der **AX-Variante** (Adapter-basierte Funktionsbausteine). Die Steuerung erfolgt über drei Softkeys (F1, F2, F3). Die Funktionsweise ist wie folgt:

- **SoftKey F1** ⟶ Zylinder 1 fährt aus.
- **SoftKey F2** ⟶ Zylinder 1 fährt ein, gleichzeitig Zylinder 2 fährt aus (Spiegelung).
- **SoftKey F3** ⟶ Zylinder 2 fährt ein.

Die digitalen Ausgänge Q1 und Q2 steuern die Endlagen der Zylinder (z. B. Ventile). Der Ablauf ist typisch für industrielle Abfolgesteuerungen mit einem „Verschieben“ des aktiven Zylinders.

Verwendete Funktionsbausteine (FBs)

Die Übung besteht aus einem SubApp-Netzwerk mit sieben Funktionsbausteinen und einem Ereignis-Splitter.

DigitalOutput_Q1, DigitalOutput_Q2

- **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
- **Parameter**:
  - `QI` = `TRUE`
  - `Output` = `Output_Q1` bzw. `Output_Q2`
- **Funktionsweise**: Stellt einen digitalen Ausgang dar. Der Ausgang wird aktiv, wenn der Eingang (über den Adapter-Port `OUT`) ein Signal erhält.

SoftKey_UP_F1, SoftKey_F2_DOWN, SoftKey_F3_DOWN

- **Typ**: `isobus::UT::io::Softkey::Softkey_IXA`
- **Parameter**:
  - `QI` = `TRUE`

  - u16ObjId = Referenz auf den entsprechenden Softkey (SoftKey_F1, SoftKey_F2, SoftKey_F3)
- **Funktionsweise**: Erzeugt ein Ereignis am Adapter-Ausgang IN, sobald die zugeordnete Taste (F1, F2, F3) gedrückt wird. Der Eingang QI aktiviert das Bauteil.

AX_FB_SR_Ausfahren_Cyl_1, AX_FB_SR_Ausfahren_Cyl_2

- **Typ**: adapter::iec61131::bistableElements::AX_FB_SR
- **Parameter**: Keine (alle Konfiguration über Adapter-Schnittstelle)
- **Verwendete interne FBs**:
  - Es handelt sich um ein **SR-Flipflop** (Set-Reset) in der AX-Adapter-Variante.
  - **Adapter-Ports**: SET1 (Setzen), RESET (Rücksetzen), Q1 (Ausgang).
- **Funktionsweise**: Ein Set-Impuls an SET1 setzt den Ausgang Q1 auf TRUE und hält diesen, bis ein Reset-Impuls an RESET erfolgt. Das Verhalten entspricht einem dominanten Set-Flipflop.

AX_SPLIT_2

- **Typ**: adapter::events::unidirectional::AX_SPLIT_2
- **Parameter**: Keine
- **Funktionsweise**: Ein Ereignis-Splitter. Ein eingehendes Ereignis am Eingang IN wird auf zwei Ausgänge (OUT1, OUT2) verteilt – beide werden gleichzeitig aktiviert. Dient zur Weiterleitung eines Tastendrucks an zwei Ziele.

Programmablauf und Verbindungen

Der Ablauf wird durch die Adapter-Verbindungen im SubApp-Netzwerk festgelegt:

1. **F1 (SoftKey_UP_F1)** → setzt AX_FB_SR_Ausfahren_Cyl_1 (über SET1).
   - Der Ausgang Q1 von Cyl_1 wird TRUE und schaltet den **DigitalOutput_Q1** ein (Zylinder 1 fährt aus).
2. **F2 (SoftKey_F2_DOWN)** → wird über AX_SPLIT_2 auf zwei Pfade verteilt:
   - **OUT1** → RESET von AX_FB_SR_Ausfahren_Cyl_1 → Zylinder 1 fährt ein (Q1 = FALSE).
   - **OUT2** → SET1 von AX_FB_SR_Ausfahren_Cyl_2 → Zylinder 2 fährt aus (Q2 = TRUE).
   - Damit wird die Spiegelung realisiert: Der aktive Zylinder wechselt von 1 zu 2.
3. **F3 (SoftKey_F3_DOWN)** → RESET von AX_FB_SR_Ausfahren_Cyl_2 → Zylinder 2 fährt ein (Q2 = FALSE).

Übersicht der Signalflüsse:

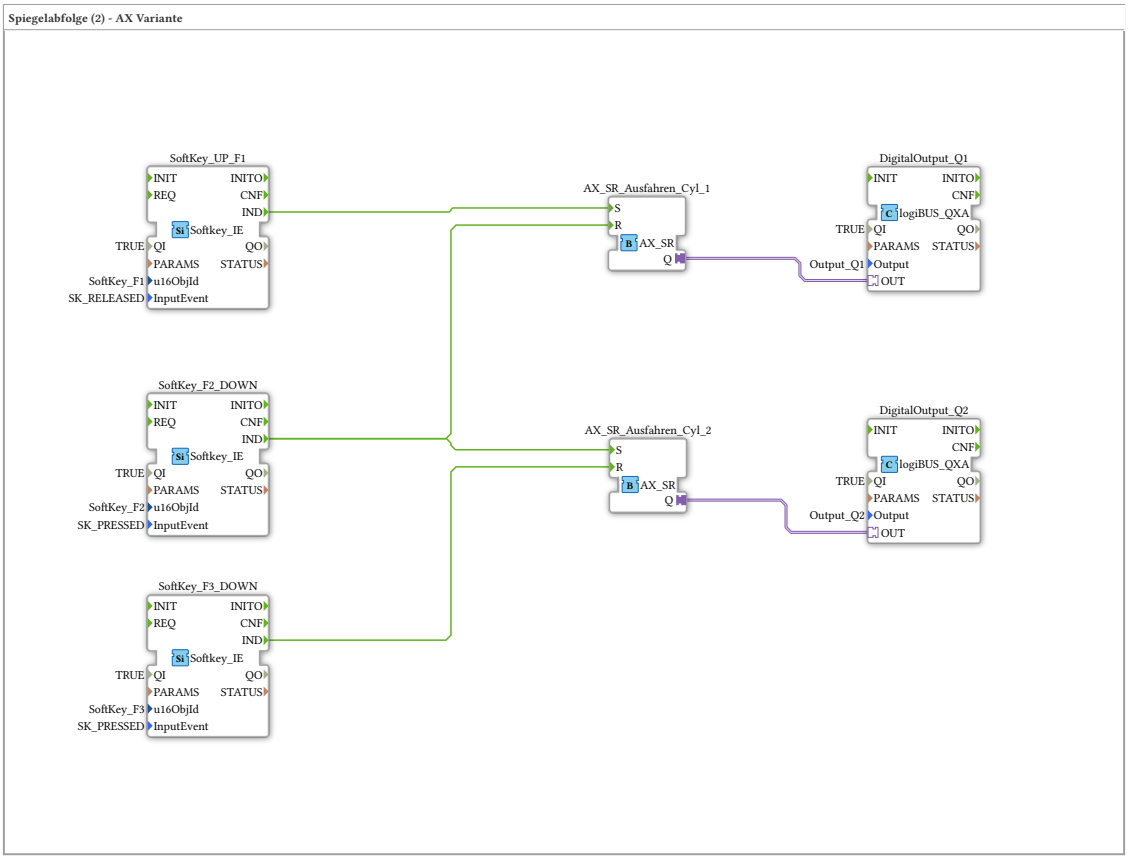
- F1 → Set Cyl_1 → Q1 aktiv
- F2 → Reset Cyl_1 + Set Cyl_2 → Q1 inaktiv, Q2 aktiv
- F3 → Reset Cyl_2 → Q2 inaktiv

Die Kommentare im Netzwerk markieren die Softkeys als „START-Knopf“ (F1) und die Endlagen der Zylinder.

Zusammenfassung

Die Übung **Uebung_022_AX** demonstriert eine einfache Spiegelabfolge für zwei Zylinder mit AX-Adapter-Bausteinen. Die Steuerung erfolgt rein ereignisbasiert über drei Softkeys. Ein SR-Flipflop pro Zylinder speichert den Zustand, während ein Ereignis-Splitter den Befehl von F2 auf zwei Ziele verteilt. Diese Struktur ist typisch für die Umsetzung von Ablaufsteuerungen in der IEC 61499 mit Adapter-Schnittstellen.

2.1.2.256 Uebung_022b_AX2: Spiegelabfolge (2) - AX Variante



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung realisiert eine **Spiegelabfolge** für zwei pneumatische Zylinder (Cyl_1 und Cyl_2) unter Verwendung von Softkeys als Bedienelemente und AX_SR-Funktionsbausteinen zur Ansteuerung der Digitalausgänge. Die Abfolge wird durch drei Tasten (F1, F2, F3) gesteuert: Mit F1 wird Zylinder 1 ausgefahren, mit F2 wird Zylinder 1 eingefahren und gleichzeitig Zylinder 2 ausgefahren, mit F3 wird Zylinder 2 eingefahren. Die Übung vermittelt den Umgang mit Set-Reset-Adapterbausteinen und deren Verknüpfung mit Ereignissen und Datenausgängen.

• – ○ • – ○ • – ○ •

Verwendete Funktionsbausteine (FBs)

Die Übung besteht aus fünf Funktionsbausteinen, die im SubApp-Netzwerk verdrahtet sind:

1. **SoftKey_UP_F1**
   - **Typ**: `isobus::UT::io::Softkey::Softkey_IE`
   - **Parameter**:
     - `QI` = `TRUE`
     - `u16ObjId` = `SoftKey_F1`
     - `InputEvent` = `SK_RELEASED` (Ereignis bei Loslassen der Taste F1)
   - **Ereignisausgang**: `IND` (wird bei Betätigung der Taste ausgelöst)
2. **SoftKey_F2_DOWN**
   - **Typ**: `Softkey_IE`
   - **Parameter**:
     - `QI` = `TRUE`
     - `u16ObjId` = `SoftKey_F2`
     - `InputEvent` = `SK_PRESSED` (Ereignis beim Drücken der Taste F2)
   - **Ereignisausgang**: `IND`

3. **SoftKey_F3_DOWN**
   - **Typ**: `Softkey_IE`
   - **Parameter**:
     - `QI` = `TRUE`
     - `u16ObjId` = `SoftKey_F3`
     - `InputEvent` = `SK_PRESSED` (Ereignis beim Drücken der Taste F3)
   - **Ereignisausgang**: `IND`
4. **AX_SR_Ausfahren_Cyl_1**
   - **Typ**: `adapter::events::unidirectional::AX_SR` (Set-Reset-Funktionsbaustein)
   - **Adapter**: unidirektional, Ausgang `Q` liefert `TRUE`, wenn gesetzt
   - **Ereigniseingänge**:
     - `S` – Setzen (Ausgang Q = TRUE)
     - `R` – Rücksetzen (Ausgang Q = FALSE)
5. **AX_SR_Ausfahren_Cyl_2**
   - **Typ**: `AX_SR` (baugleich zu Cyl_1)
   - **Ereigniseingänge**:
     - `S` – Setzen
     - `R` – Rücksetzen
6. **DigitalOutput_Q1**
   - **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
   - **Parameter**:
     - `QI` = `TRUE` (Ausgang freigegeben)
     - `Output` = `Output_Q1` (physischer Ausgang)
   - **Adaptereingang**: `OUT` – steuert den Ausgang bei `TRUE`
7. **DigitalOutput_Q2**
   - **Typ**: `logiBUS_QXA`
   - **Parameter**:
     - `QI` = `TRUE`
     - `Output` = `Output_Q2`
   - **Adaptereingang**: `OUT`

Sub-Bausteine

Keine separaten Unter-Bausteine vorhanden. Die gesamte Logik ist direkt im SubApp-Netzwerk realisiert.

• – ◦ • – ◦ • – ◦ •

Programmablauf und Verbindungen

Die Steuerung folgt einer festen Abfolge:

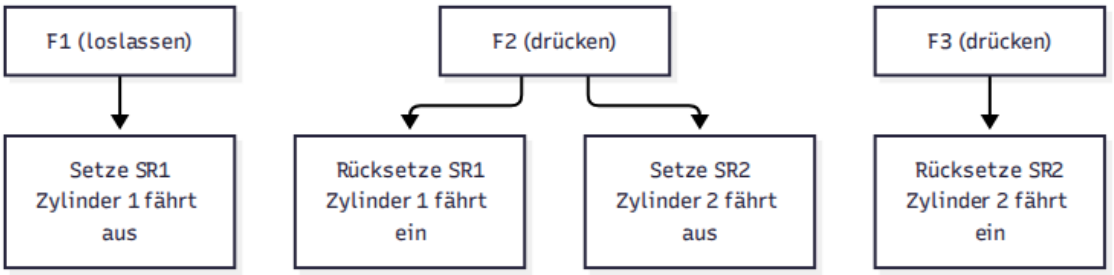
1. **Taste F1 loslassen** ⟶ Ereignis von `SoftKey_UP_F1.IND` ⟶ Setzt `AX_SR_Ausfahren_Cyl_1.S` ⟶ **Zylinder 1 fährt aus** (DigitalOutput_Q1 = TRUE).
2. **Taste F2 drücken** ⟶ Ereignis von `SoftKey_F2_DOWN.IND` ⟶ Wird an zwei Ziele verteilt:
   - `AX_SR_Ausfahren_Cyl_1.R` ⟶ **Zylinder 1 fährt ein** (Q1 = FALSE).
   - `AX_SR_Ausfahren_Cyl_2.S` ⟶ **Zylinder 2 fährt aus** (Q2 = TRUE).
3. **Taste F3 drücken** ⟶ Ereignis von `SoftKey_F3_DOWN.IND` ⟶ Setzt `AX_SR_Ausfahren_Cyl_2.R` ⟶ **Zylinder 2 fährt ein** (Q2 = FALSE).

Die Verbindungen im Detail:

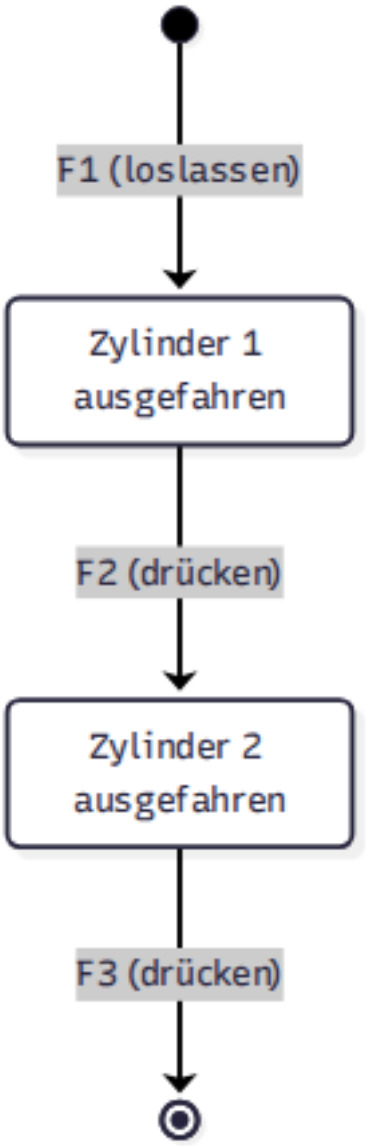
Von	Nach	Typ
`SoftKey_UP_F1.IND`	`AX_SR_Ausfahren_Cyl_1.S`	Ereignis
`SoftKey_F2_DOWN.IND`	`AX_SR_Ausfahren_Cyl_1.R`	Ereignis
`SoftKey_F2_DOWN.IND`	`AX_SR_Ausfahren_Cyl_2.S`	Ereignis

SoftKey_F3_DOWN.IND	AX_SR_Ausfahren_Cyl_2.R	Ereignis
AX_SR_Ausfahren_Cyl_1.Q	DigitalOutput_Q1.OUT	Adapter
AX_SR_Ausfahren_Cyl_2.Q	DigitalOutput_Q2.OUT	Adapter

Ablaufdiagramm (vereinfacht):



Als Zustandsdiagramm sieht die Spiegelabfolge wie folgt aus:



Lernziele:

- Verwendung von Set-Reset-Funktionsbausteinen (AX_SR) in 4diac.
- Verknüpfung mehrerer Ereignisquellen mit einem Ziel (Fan-Out).
- Steuerung von Digitalausgängen über Adapter.
- Erstellen einer einfachen Ablaufsteuerung mit Tasteneingaben.

Schwierigkeitsgrad: Einfach **Vorkenntnisse:** Grundlagen der IEC 61499, Umgang mit dem 4diac-IDE

Hinweise zum Starten: Die Übung ist als SubApp-Typ vorgefertigt. Einbinden in ein geeignetes Projekt und mit einem Laufzeitsystem (z. B. FORTE) ausführen. Die physischen Ausgänge Q1 und Q2 müssen entsprechend den verwendeten Komponenten (z. B. Ventile) angeschlossen sein.

• – ◦ • – ◦ • – ◦ •

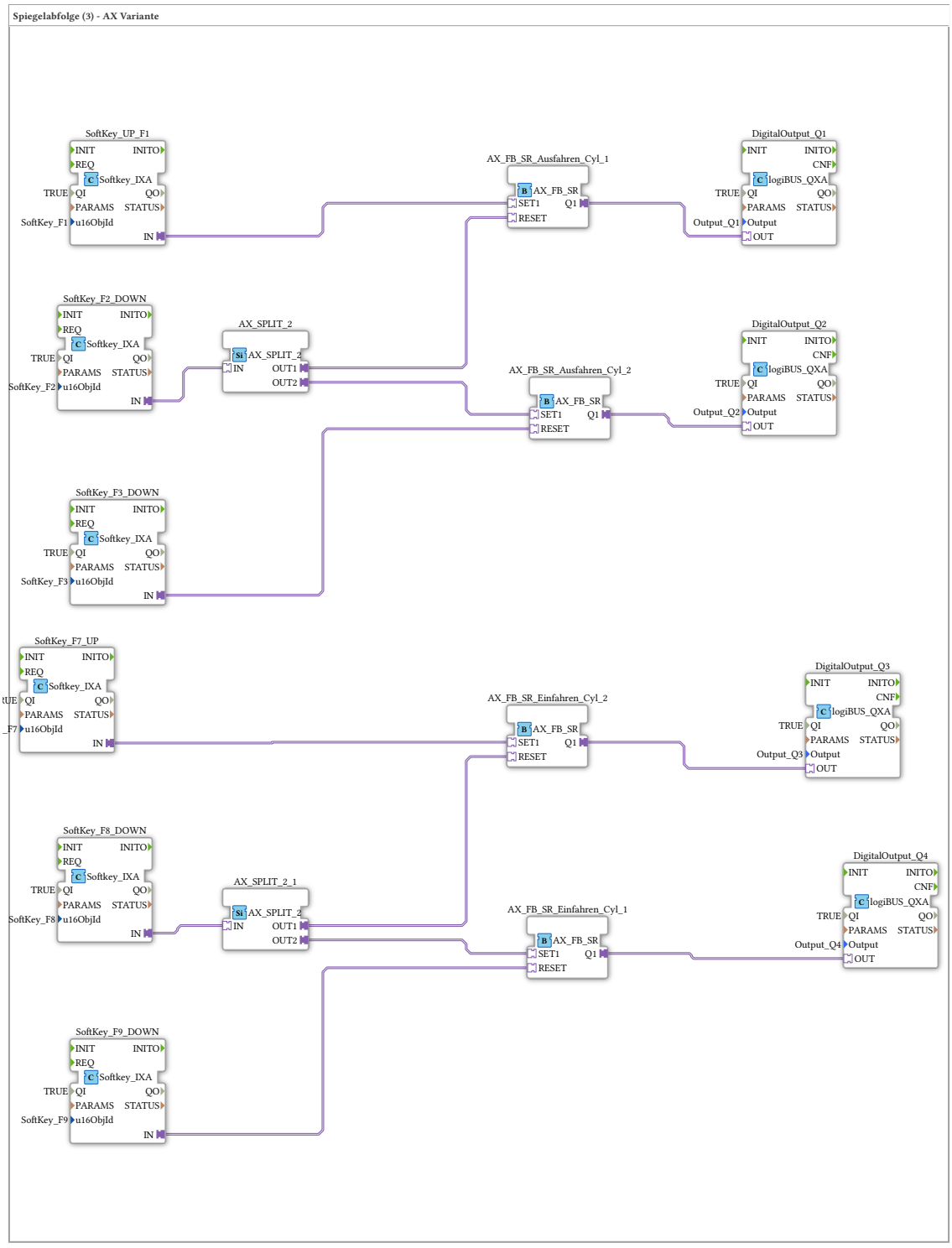
Zusammenfassung

Die Übung **Uebung_022b_AX2** demonstriert eine zweistufige Ablaufsteuerung für zwei Zylinder mithilfe von AX_SR-Funktionsbausteinen. Durch die geschickte Verteilung der Ereignisse (F2 löst sowohl Rücksetzen des ersten als auch Setzen des zweiten Zylinders aus) wird eine „Spiegelabfolge“ realisiert. Der Aufbau ist einfach, erweiterbar und zeigt die Grundprinzipien der ereignisgesteuerten Automatisierung in 4diac. Die Verwendung von Adaptern statt direkter Datenverbindungen sorgt für eine saubere Trennung von Steuerlogik und Aktorik.

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

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

2.1.2.257 Uebung_023_AX: Spiegelabfolge (3) - AX Variante



• — ○ • — ○ • — ○ •

Einleitung

Diese Übung realisiert eine **Spiegelabfolge** für zwei doppeltwirkende Zylinder. Ziel ist es, eine sequenzielle Steuerung für das Aus- und Einfahren zweier Zylinder mithilfe von AX-Funktionsbausteinen (SR-Bistabile) und Softkeys aufzubauen. Die Steuerung wird über Softkeys am Terminal bedient, die digitalen Ausgänge steuern die Stellglieder (z. B. Ventile) an.

Die Abfolge ist in zwei Teilbereiche gegliedert:

- **Ausfahren der Zylinder** (Zylinder 1 und 2)
- **Einfahren der Zylinder** (Zylinder 2 und 1)

Durch die AX_SPLIT-Adapter werden Ereignisse auf mehrere Pfade verteilt, sodass die Befehlsketten in der korrekten Reihenfolge ablaufen.

Verwendete Funktionsbausteine (FBs)

Es werden keine weiteren Sub-Bausteine (SubAppTypes) verwendet – alle Funktionsbausteine sind direkt im Netzwerk eingebunden.

Eingangsbausteine (Softkeys)

- **SoftKey_F1 (UP)** Typ: `isobus::UT::io::Softkey::Softkey_IXA` Parameter:
  - `QI` = TRUE
  - `u16ObjId` = SoftKey_F1 Funktion: Starttaste **Ausfahren** für Zylinder 1.
- **SoftKey_F2 (DOWN)** Typ: `isobus::UT::io::Softkey::Softkey_IXA` Parameter:
  - `QI` = TRUE
  - `u16ObjId` = SoftKey_F2 Funktion: Löst über `AX_SPLIT_2` das Zurücksetzen von Zylinder 1 Ausfahren und gleichzeitiges Setzen von Zylinder 2 Ausfahren aus (Spiegelung).
- **SoftKey_F3 (DOWN)** Typ: `isobus::UT::io::Softkey::Softkey_IXA` Parameter:
  - `QI` = TRUE
  - `u16ObjId` = SoftKey_F3 Funktion: Reset (Einfahren) für **Ausfahren Zylinder 2**.
- **SoftKey_F7 (UP)** Typ: `isobus::UT::io::Softkey::Softkey_IXA` Parameter:
  - `QI` = TRUE
  - `u16ObjId` = SoftKey_F7 Funktion: Starttaste **Einfahren** für Zylinder 2.
- **SoftKey_F8 (DOWN)** Typ: `isobus::UT::io::Softkey::Softkey_IXA` Parameter:
  - `QI` = TRUE
  - `u16ObjId` = SoftKey_F8 Funktion: Löst über `AX_SPLIT_3` das Zurücksetzen von Zylinder 2 Einfahren und gleichzeitiges Setzen von Zylinder 1 Einfahren aus (Spiegelung).
- **SoftKey_F9 (DOWN)** Typ: `isobus::UT::io::Softkey::Softkey_IXA` Parameter:
  - `QI` = TRUE
  - `u16ObjId` = SoftKey_F9 Funktion: Reset (Ausfahren) für **Einfahren Zylinder 1**.

Verarbeitungsbausteine

- **AX_FB_SR_Ausfahren_Cyl_1** Typ: `adapter::iec61131::bistableElements::AX_FB_SR` Parameter: keine (keine weiteren Parameter in der XML) Funktion: SR-Bistabil für den Ausfahrbefehl von Zylinder 1. Setzen durch SoftKey_F1, Zurücksetzen durch `AX_SPLIT_2.OUT1`.
- **AX_FB_SR_Ausfahren_Cyl_2** Typ: `adapter::iec61131::bistableElements::AX_FB_SR` Parameter: keine Funktion: SR-Bistabil für den Ausfahrbefehl von Zylinder 2. Setzen durch `AX_SPLIT_2.OUT2`, Zurücksetzen durch SoftKey_F3.
- **AX_FB_SR_Einfahren_Cyl_2** Typ: `adapter::iec61131::bistableElements::AX_FB_SR` Parameter: keine Funktion: SR-Bistabil für den Einfahrbefehl von Zylinder 2. Setzen durch SoftKey_F7, Zurücksetzen durch `AX_SPLIT_3.OUT1`.
- **AX_FB_SR_Einfahren_Cyl_1** Typ: `adapter::iec61131::bistableElements::AX_FB_SR` Parameter: keine Funktion: SR-Bistabil für den Einfahrbefehl von Zylinder 1. Setzen durch `AX_SPLIT_3.OUT2`, Zurücksetzen durch SoftKey_F9.

Ereignisverteilung

- **AX_SPLIT_2** Typ: `adapter::events::unidirectional::AX_SPLIT_2` Parameter: keine Funktion: Verteilt das Ereignis von SoftKey_F2 auf zwei Ausgänge:
  - OUT1 ⟶ RESET von AX_FB_SR_Ausfahren_Cyl_1
  - OUT2 ⟶ SET von AX_FB_SR_Ausfahren_Cyl_2
- **AX_SPLIT_3** Typ: `adapter::events::unidirectional::AX_SPLIT_2` Parameter: keine Funktion: Verteilt das Ereignis von SoftKey_F8 auf zwei Ausgänge:

- OUT1 ⟶ RESET von AX_FB_SR_Einfahren_Cyl_2
- OUT2 ⟶ SET von AX_FB_SR_Einfahren_Cyl_1

Ausgangsbausteine

- **DigitalOutput_Q1** Typ: `logiBUS::io::DQ::logiBUS_QXA` Parameter:
  - `QI` = TRUE
  - `Output` = Output_Q1 Funktion: Gibt den Ausgang Q1 (Ausfahren Zylinder 1) aus. Gesteuert von `AX_FB_SR_Ausfahren_Cyl_1.Q1`.
- **DigitalOutput_Q2** Typ: `logiBUS::io::DQ::logiBUS_QXA` Parameter:
  - `QI` = TRUE
  - `Output` = Output_Q2 Funktion: Gibt den Ausgang Q2 (Ausfahren Zylinder 2) aus. Gesteuert von `AX_FB_SR_Ausfahren_Cyl_2.Q1`.
- **DigitalOutput_Q3** Typ: `logiBUS::io::DQ::logiBUS_QXA` Parameter:
  - `QI` = TRUE
  - `Output` = Output_Q3 Funktion: Gibt den Ausgang Q3 (Einfahren Zylinder 2) aus. Gesteuert von `AX_FB_SR_Einfahren_Cyl_2.Q1`.
- **DigitalOutput_Q4** Typ: `logiBUS::io::DQ::logiBUS_QXA` Parameter:
  - `QI` = TRUE
  - `Output` = Output_Q4 Funktion: Gibt den Ausgang Q4 (Einfahren Zylinder 1) aus. Gesteuert von `AX_FB_SR_Einfahren_Cyl_1.Q1`.

Programmablauf und Verbindungen

Ausfahrsequenz

1. **Start Ausfahren Zylinder 1** Durch Drücken von **SoftKey_F1** (UP) wird der Eingang SET1 von `AX_FB_SR_Ausfahren_Cyl_1` gesetzt. Dadurch wird `Q1` aktiv und der Ausgang Q1 schaltet.
2. **Spiegelung auf Zylinder 2** Durch Drücken von **SoftKey_F2** (DOWN) wird ein Ereignis an `AX_SPLIT_2` gesendet.
   - Der Ausgang OUT1 von `AX_SPLIT_2` resettet das SR-Bistabil `AX_FB_SR_Ausfahren_Cyl_1` ⟶ Q1 wird inaktiv, Zylinder 1 fährt nicht mehr aus.
   - Der Ausgang OUT2 setzt gleichzeitig `AX_FB_SR_Ausfahren_Cyl_2` ⟶ Q2 aktiv, Zylinder 2 fährt aus.
3. **Reset Zylinder 2** Durch Drücken von **SoftKey_F3** (DOWN) wird `AX_FB_SR_Ausfahren_Cyl_2` zurückgesetzt ⟶ Q2 inaktiv.

Einfahrsequenz (in umgekehrter Reihenfolge)

1. **Start Einfahren Zylinder 2** Durch Drücken von **SoftKey_F7** (UP) wird `AX_FB_SR_Einfahren_Cyl_2` gesetzt ⟶ Q3 aktiv.
2. **Spiegelung auf Zylinder 1** Durch Drücken von **SoftKey_F8** (DOWN) wird `AX_SPLIT_3` angesteuert:
   - OUT1 resettet `AX_FB_SR_Einfahren_Cyl_2` ⟶ Q3 inaktiv.
   - OUT2 setzt `AX_FB_SR_Einfahren_Cyl_1` ⟶ Q4 aktiv.
3. **Reset Zylinder 1** Durch Drücken von **SoftKey_F9** (DOWN) wird `AX_FB_SR_Einfahren_Cyl_1` zurückgesetzt ⟶ Q4 inaktiv.

Verbindungsstruktur (AdapterConnections)

Die Adapterverbindungen sind im Netzwerk wie folgt verknüpft:

- `SoftKey_UP_F1.IN` ⟶ `AX_FB_SR_Ausfahren_Cyl_1.SET1`
- `SoftKey_F2_DOWN.IN` ⟶ `AX_SPLIT_2.IN`
- `AX_SPLIT_2.OUT1` ⟶ `AX_FB_SR_Ausfahren_Cyl_1.RESET`
- `AX_SPLIT_2.OUT2` ⟶ `AX_FB_SR_Ausfahren_Cyl_2.SET1`
- `SoftKey_F3_DOWN.IN` ⟶ `AX_FB_SR_Ausfahren_Cyl_2.RESET`
- `SoftKey_F7_UP.IN` ⟶ `AX_FB_SR_Einfahren_Cyl_2.SET1`

- `SoftKey_F8_DOWN.IN` ⟶ `AX_SPLIT_3.IN`
- `AX_SPLIT_3.OUT1` ⟶ `AX_FB_SR_Einfahren_Cyl_2.RESET`
- `AX_SPLIT_3.OUT2` ⟶ `AX_FB_SR_Einfahren_Cyl_1.SET1`
- `SoftKey_F9_DOWN.IN` ⟶ `AX_FB_SR_Einfahren_Cyl_1.RESET`
- `AX_FB_SR_Ausfahren_Cyl_1.Q1` ⟶ `DigitalOutput_Q1.OUT`
- `AX_FB_SR_Ausfahren_Cyl_2.Q1` ⟶ `DigitalOutput_Q2.OUT`
- `AX_FB_SR_Einfahren_Cyl_2.Q1` ⟶ `DigitalOutput_Q3.OUT`
- `AX_FB_SR_Einfahren_Cyl_1.Q1` ⟶ `DigitalOutput_Q4.OUT`

Lernziele

- Verständnis der Spiegelabfolge (Abfolgebildung mit Rücksetzen und Setzen in einem Schritt)
- Anwendung von **AX_FB_SR** (bistabile Elemente nach IEC 61131)
- Einsatz von **AX_SPLIT_2** zur Ereignisverteilung
- Verknüpfung von Softkey-Eingaben mit digitalen Ausgängen über logiBUS
- Einüben des vierstufigen Ablaufs: Starten – Spiegeln – Stoppen – Rücksetzen

Hinweise

- Die Übung erfordert Grundkenntnisse in der 4diac-IDE und im Umgang mit Adapterverbindungen.
- Alle Softkeys sind als **DOWN**- bzw. **UP**-Tasten definiert; die Zuordnung erfolgt über die Objekt-ID (z. B. `SoftKey_F1`).
- Die digitalen Ausgänge `Output_Q1` bis `Output_Q4` müssen den realen Ventilen oder Anzeigen zugeordnet sein.

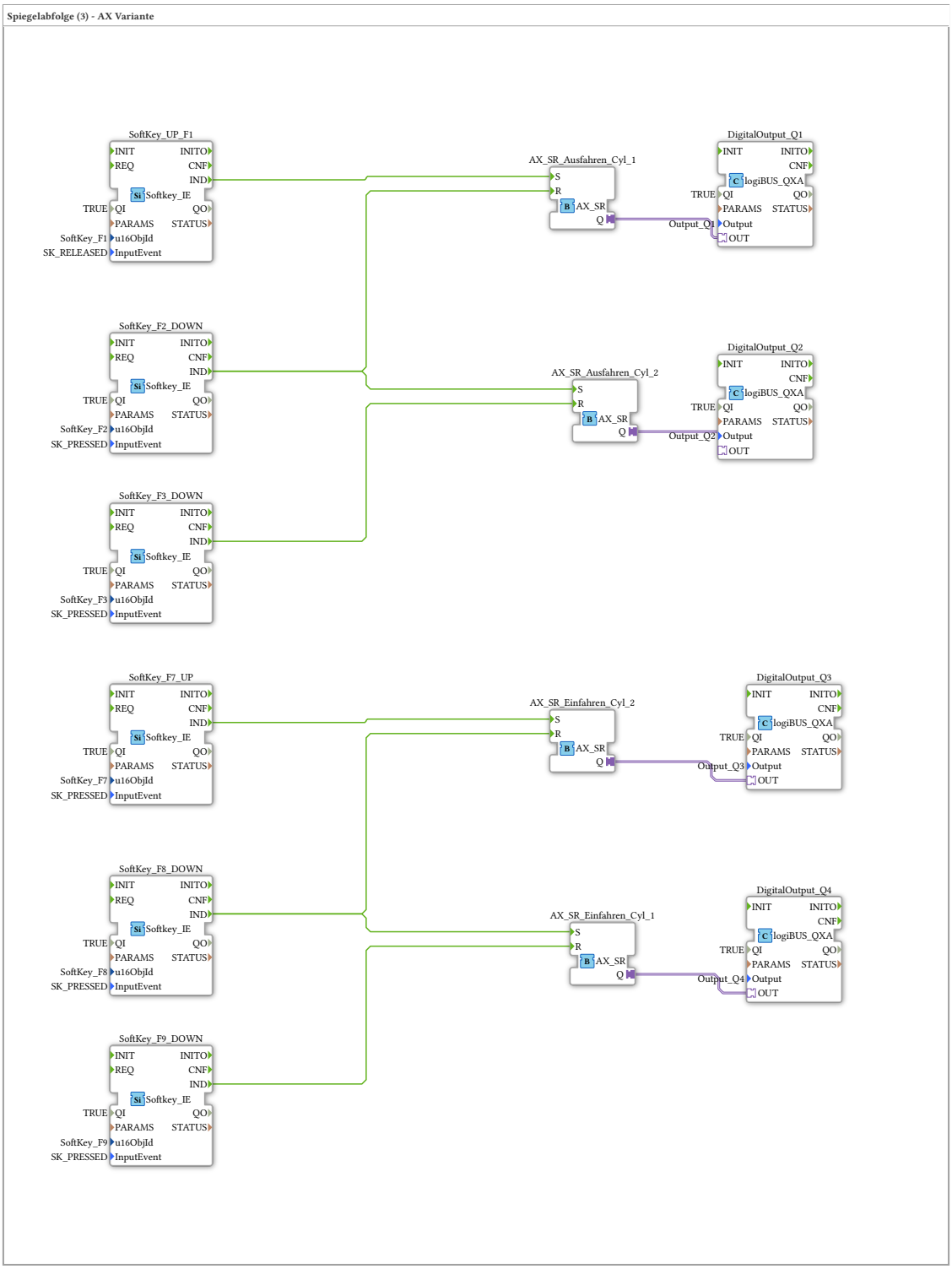
Zusammenfassung

Die Übung „Spiegelabfolge (3) – AX Variante“ demonstriert eine sequenzielle Steuerung für zwei Zylinder mit Hilfe von SR-Speicherbausteinen und Ereignisverteilern. Durch die AX_SPLIT-Adapter werden beim Drücken eines Softkeys gleichzeitig ein SR zurückgesetzt und ein weiteres SR gesetzt – dadurch wird die „Spiegelung“ der Bewegung realisiert. Das Programm besteht aus einer klaren Trennung von Ausfahr- und Einfahrsequenz und ermöglicht ein schrittweises Verständnis der Signalverkettung in der Automatisierungstechnik. Der Einsatz der logiBUS-Ausgänge verbindet die Steuerung direkt mit der Peripherie.

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

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

2.1.2.258 Uebung_023b_AX2: Spiegelabfolge (3) - AX Variante



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung realisiert eine **Spiegelabfolge** für zwei doppeltwirkende Zylinder (Zylinder 1 und Zylinder 2) mittels AX-SR-Bausteinen (unidirektionale Adapter). Die Steuerung erfolgt über acht Softkeys (F1, F2, F3, F7, F8, F9) in Verbindung mit vier digitalen Ausgängen (Q1...Q4). Das Verhalten ist symmetrisch aufgebaut:

- **Ausfahren** von Zylinder 1 und 2 wird durch Softkeys und AX-SR-Bausteine gesteuert.

- **Einfahren** von Zylinder 1 und 2 erfolgt analog mit eigenen Softkeys.

Der SubAppType besitzt keine eigenen Ein-/Ausgangs-Schnittstellen, sondern kommuniziert ausschließlich über die eingebundenen System-Bausteine mit der Hardware.

Verwendete Funktionsbausteine (FBs)

Bausteinname	Typ	Beschreibung
SoftKey_UP_F1	`isobus::UT::io::Softkey::Softkey_IE`	Softkey F1 – ausgelöst bei **Taste loslassen** (SK_RELEASED)
SoftKey_F2_DOWN	`isobus::UT::io::Softkey::Softkey_IE`	Softkey F2 – ausgelöst bei **Taste drücken** (SK_PRESSED)
SoftKey_F3_DOWN	`isobus::UT::io::Softkey::Softkey_IE`	Softkey F3 – ausgelöst bei **Taste drücken** (SK_PRESSED)
SoftKey_F7_UP	`isobus::UT::io::Softkey::Softkey_IE`	Softkey F7 – ausgelöst bei **Taste drücken** (SK_PRESSED)
SoftKey_F8_DOWN	`isobus::UT::io::Softkey::Softkey_IE`	Softkey F8 – ausgelöst bei **Taste drücken** (SK_PRESSED)
SoftKey_F9_DOWN	`isobus::UT::io::Softkey::Softkey_IE`	Softkey F9 – ausgelöst bei **Taste drücken** (SK_PRESSED)
AX_SR_Ausfahren_Cyl_1	`adapter::events::unidirectional::AX_SR`	Set-Reset-Baustein für Ausfahren von Zylinder 1 (S: Set, R: Reset)
AX_SR_Ausfahren_Cyl_2	`adapter::events::unidirectional::AX_SR`	Set-Reset-Baustein für Ausfahren von Zylinder 2
AX_SR_Einfahren_Cyl_1	`adapter::events::unidirectional::AX_SR`	Set-Reset-Baustein für Einfahren von Zylinder 1
AX_SR_Einfahren_Cyl_2	`adapter::events::unidirectional::AX_SR`	Set-Reset-Baustein für Einfahren von Zylinder 2
DigitalOutput_Q1	`logiBUS::io::DQ::logiBUS_QXA`	Digitaler Ausgang Q1 (aktiv wenn AX_SR_Ausfahren_Cyl_1.Q gesetzt)
DigitalOutput_Q2	`logiBUS::io::DQ::logiBUS_QXA`	Digitaler Ausgang Q2 (aktiv wenn AX_SR_Ausfahren_Cyl_2.Q gesetzt)
DigitalOutput_Q3	`logiBUS::io::DQ::logiBUS_QXA`	Digitaler Ausgang Q3 (aktiv wenn AX_SR_Einfahren_Cyl_2.Q gesetzt)
DigitalOutput_Q4	`logiBUS::io::DQ::logiBUS_QXA`	Digitaler Ausgang Q4 (aktiv wenn AX_SR_Einfahren_Cyl_1.Q gesetzt)

Parameter der Bausteine

Alle `Softkey_IE`-Bausteine sind mit folgenden Parametern konfiguriert:

- **QI** = TRUE
- **u16ObjId** = jeweilige Softkey-Konstante (z.B. `SoftKey_F1`)
- **InputEvent** = Auslöser (SK_RELEASED oder SK_PRESSED)

Alle `logiBUS_QXA`-Bausteine sind mit folgenden Parametern konfiguriert:

- **QI** = TRUE
- **Output** = jeweiliger Ausgang (Output_Q1 … Output_Q4)

Die AX_SR-Bausteine haben keine Parameter, sie erhalten Ereignisse über die Eingänge S (Set) und R (Reset) und geben den Zustand über den Adapter-Ausgang Q weiter.

Programmablauf und Verbindungen

Die Steuerung ist in zwei voneinander unabhängige Zyklen gegliedert:

Ausfahren der Zylinder

1. **Zylinder 1 ausfahren**
   - Softkey **F1** loslassen → Ereignis am IND-Ausgang von SoftKey_UP_F1 wird an den S-Eingang von AX_SR_Ausfahren_Cyl_1 geleitet.
   - Softkey **F2** drücken → Ereignis von SoftKey_F2_DOWN.IND wird an den R-Eingang von AX_SR_Ausfahren_Cyl_1 geleitet.
   - Der Zustand Q von AX_SR_Ausfahren_Cyl_1 wird über den Adapter an den OUT-Eingang von DigitalOutput_Q1 weitergegeben → **Ausgang Q1** schaltet.
2. **Zylinder 2 ausfahren**
   - Softkey **F2** drücken → Ereignis von SoftKey_F2_DOWN.IND wird an den S-Eingang von AX_SR_Ausfahren_Cyl_2 geleitet.
   - Softkey **F3** drücken → Ereignis von SoftKey_F3_DOWN.IND wird an den R-Eingang von AX_SR_Ausfahren_Cyl_2 geleitet.
   - Der Zustand Q von AX_SR_Ausfahren_Cyl_2 wird an DigitalOutput_Q2 weitergegeben → **Ausgang Q2** schaltet.

Einfahren der Zylinder

1. **Zylinder 1 einfahren**
   - Softkey **F8** drücken → Ereignis von SoftKey_F8_DOWN.IND wird an den S-Eingang von AX_SR_Einfahren_Cyl_1 geleitet.
   - Softkey **F9** drücken → Ereignis von SoftKey_F9_DOWN.IND wird an den R-Eingang von AX_SR_Einfahren_Cyl_1 geleitet.
   - Der Zustand Q von AX_SR_Einfahren_Cyl_1 wird an DigitalOutput_Q4 weitergegeben → **Ausgang Q4** schaltet.
2. **Zylinder 2 einfahren**
   - Softkey **F7** drücken → Ereignis von SoftKey_F7_UP.IND wird an den S-Eingang von AX_SR_Einfahren_Cyl_2 geleitet.
   - Softkey **F8** drücken → Ereignis von SoftKey_F8_DOWN.IND wird an den R-Eingang von AX_SR_Einfahren_Cyl_2 geleitet.
   - Der Zustand Q von AX_SR_Einfahren_Cyl_2 wird an DigitalOutput_Q3 weitergegeben → **Ausgang Q3** schaltet.

Grafische Anordnung der Kommentare (zur Orientierung)

- **START-Knopf Ausfahren** (nahe F1)
- **Endlage Ausfahren_Cyl_1** (nahe F2)
- **Endlage Ausfahren_Cyl_2** (nahe F3)
- **START-Knopf Einfahren** (nahe F7)
- **Endlage Einfahren_Cyl_2** (nahe F8)
- **Endlage Einfahren_Cyl_1** (nahe F9)

Zusammenfassung

Die Übung *Uebung_023b_AX2* demonstriert die Verwendung von **AX-SR-Adaptern** zur Ansteuerung zweier doppeltwirkender Zylinder über Softkeys. Durch die getrennte Set-Reset-Logik für Aus- und Einfahren jedes Zylinders wird eine Spiegelabfolge realisiert, bei der die Richtungswechsel asynchron durch unterschiedliche Tasten ausgelöst werden. Die Steuerung ist hardwarenah über die logiBUS-Digitalausgänge angebunden und kann direkt in einer 4diac-IDE-Umgebung getestet werden.

Lernziele:

- Verständnis von AX-SR-Bausteinen (Set-Reset mit unidirektionalen Adaptern)
- Ereignisgesteuerte Verknüpfung von Softkeys mit Aktoren
- Strukturierte Programmierung von Zylindersteuerungen in 4diac

Voraussetzungen:

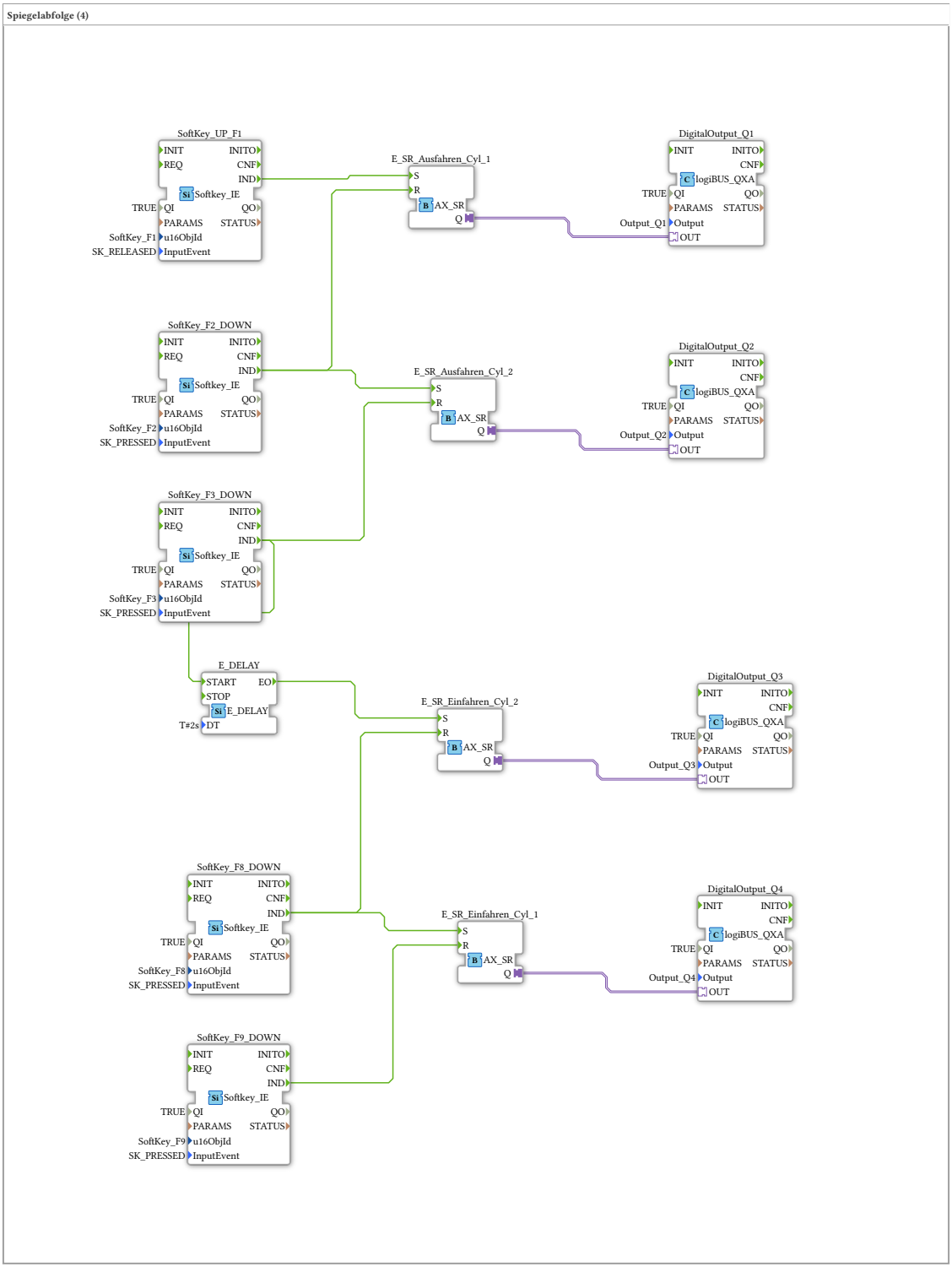
- Grundkenntnisse in 4diac und IEC 61499
- Verfügbarkeit der Bibliotheken `isobus`, `logiBUS` und `adapter::events::unidirectional`

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

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

2.1.2.259 Uebung_024_AX: Spiegelabfolge (4)

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_024_AX (Spiegelabfolge (4)).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Spiegelabfolge (4)**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_024_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.

  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
- **SoftKey_UP_F1**: Instanz des Typs isobus::UT::io::Softkey::Softkey_IE.
  - Parameter QI = TRUE
  - Parameter u16ObjId = SoftKey_F1
  - Parameter InputEvent = SK_RELEASED
- **E_SR_Ausfahren_Cyl_1**: Instanz des Typs adapter::events::unidirectional::AX_SR.
- **SoftKey_F2_DOWN**: Instanz des Typs isobus::UT::io::Softkey::Softkey_IE.
  - Parameter QI = TRUE
  - Parameter u16ObjId = SoftKey_F2
  - Parameter InputEvent = SK_PRESSED
- **DigitalOutput_Q2**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q2
- **E_SR_Ausfahren_Cyl_2**: Instanz des Typs adapter::events::unidirectional::AX_SR.
- **SoftKey_F3_DOWN**: Instanz des Typs isobus::UT::io::Softkey::Softkey_IE.
  - Parameter QI = TRUE
  - Parameter u16ObjId = SoftKey_F3
  - Parameter InputEvent = SK_PRESSED
- **SoftKey_F9_DOWN**: Instanz des Typs isobus::UT::io::Softkey::Softkey_IE.
  - Parameter QI = TRUE
  - Parameter u16ObjId = SoftKey_F9
  - Parameter InputEvent = SK_PRESSED
- **E_SR_Einfahren_Cyl_2**: Instanz des Typs adapter::events::unidirectional::AX_SR.
- **DigitalOutput_Q3**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q3
- **DigitalOutput_Q4**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q4
- **E_SR_Einfahren_Cyl_1**: Instanz des Typs adapter::events::unidirectional::AX_SR.
- **SoftKey_F8_DOWN**: Instanz des Typs isobus::UT::io::Softkey::Softkey_IE.
  - Parameter QI = TRUE
  - Parameter u16ObjId = SoftKey_F8
  - Parameter InputEvent = SK_PRESSED
- **E_DELAY**: Instanz des Typs iec61499::events::E_DELAY.
  - Parameter DT = T#2s

Verbindungen und Schnittstellen

Adapterverbindungen:

- E_SR_Ausfahren_Cyl_1.Q -> DigitalOutput_Q1.OUT
- E_SR_Ausfahren_Cyl_2.Q -> DigitalOutput_Q2.OUT
- E_SR_Einfahren_Cyl_1.Q -> DigitalOutput_Q4.OUT
- E_SR_Einfahren_Cyl_2.Q -> DigitalOutput_Q3.OUT

Ereignisverbindungen:

- SoftKey_UP_F1.IND -> E_SR_Ausfahren_Cyl_1.S
- SoftKey_F2_DOWN.IND -> E_SR_Ausfahren_Cyl_1.R
- SoftKey_F2_DOWN.IND -> E_SR_Ausfahren_Cyl_2.S
- SoftKey_F3_DOWN.IND -> E_SR_Ausfahren_Cyl_2.R
- SoftKey_F8_DOWN.IND -> E_SR_Einfahren_Cyl_1.S
- SoftKey_F9_DOWN.IND -> E_SR_Einfahren_Cyl_1.R
- SoftKey_F8_DOWN.IND -> E_SR_Einfahren_Cyl_2.R
- E_DELAY.EO -> E_SR_Einfahren_Cyl_2.S
- SoftKey_F3_DOWN.IND -> E_DELAY.START

Hinweise aus dem Modell

START-Knopf Ausfahren Endlage Ausfahren_Cyl_1 Endlage Ausfahren_Cyl_2 Ausfahren_Cyl_1 Ausfahren_Cyl_2 Einfahren Zeit gesteuert Endlage Einfahren_Cyl_1 Einfahren_Cyl_2 Endlage Einfahren_Cyl_2 Einfahren_Cyl_1

Programmablauf und Funktionsweise

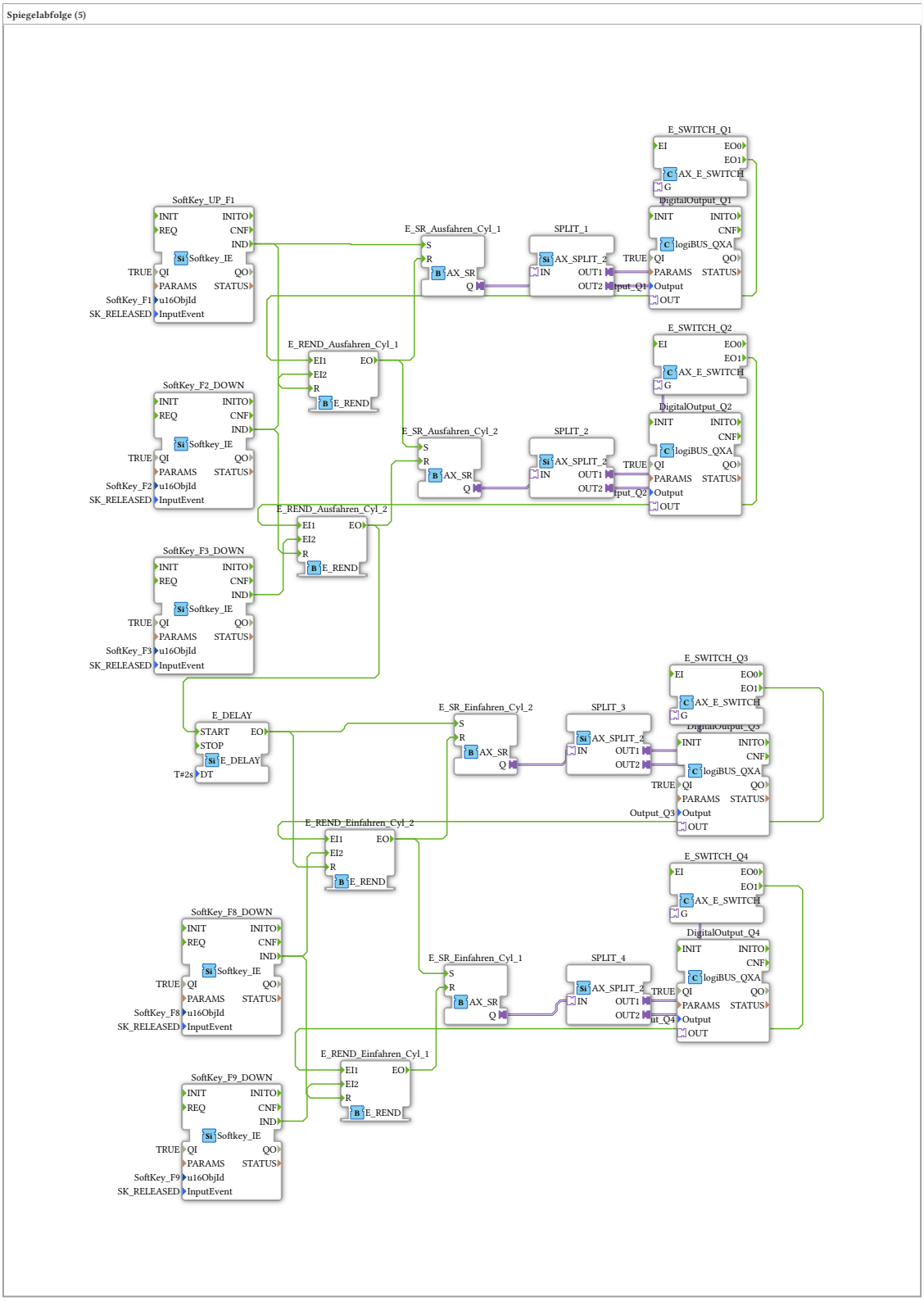
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_024_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.260 Uebung_025_AX: Spiegelabfolge (5)

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_025_AX (Spiegelabfolge (5)).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Spiegelabfolge (5)**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_025_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
- **SoftKey_UP_F1**: Instanz des Typs isobus::UT::io::Softkey::Softkey_IE.
  - Parameter QI = TRUE
  - Parameter u16ObjId = SoftKey_F1
  - Parameter InputEvent = SK_RELEASED
- **E_SR_Ausfahren_Cyl_1**: Instanz des Typs adapter::events::unidirectional::AX_SR.
- **SoftKey_F2_DOWN**: Instanz des Typs isobus::UT::io::Softkey::Softkey_IE.
  - Parameter QI = TRUE
  - Parameter u16ObjId = SoftKey_F2
  - Parameter InputEvent = SK_RELEASED
- **DigitalOutput_Q2**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q2
- **E_SR_Ausfahren_Cyl_2**: Instanz des Typs adapter::events::unidirectional::AX_SR.
- **SoftKey_F3_DOWN**: Instanz des Typs isobus::UT::io::Softkey::Softkey_IE.
  - Parameter QI = TRUE
  - Parameter u16ObjId = SoftKey_F3
  - Parameter InputEvent = SK_RELEASED
- **SoftKey_F9_DOWN**: Instanz des Typs isobus::UT::io::Softkey::Softkey_IE.
  - Parameter QI = TRUE
  - Parameter u16ObjId = SoftKey_F9
  - Parameter InputEvent = SK_RELEASED
- **E_SR_Einfahren_Cyl_2**: Instanz des Typs adapter::events::unidirectional::AX_SR.
- **DigitalOutput_Q3**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q3
- **DigitalOutput_Q4**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q4
- **E_SR_Einfahren_Cyl_1**: Instanz des Typs adapter::events::unidirectional::AX_SR.
- **SoftKey_F8_DOWN**: Instanz des Typs isobus::UT::io::Softkey::Softkey_IE.
  - Parameter QI = TRUE
  - Parameter u16ObjId = SoftKey_F8
  - Parameter InputEvent = SK_RELEASED
- **E_DELAY**: Instanz des Typs iec61499::events::E_DELAY.
  - Parameter DT = T#2s
- **E_REND_Ausfahren_Cyl_1**: Instanz des Typs iec61499::events::E_REND.
- **E_REND_Ausfahren_Cyl_2**: Instanz des Typs iec61499::events::E_REND.
- **E_REND_Einfahren_Cyl_2**: Instanz des Typs iec61499::events::E_REND.
- **E_REND_Einfahren_Cyl_1**: Instanz des Typs iec61499::events::E_REND.
- **E_SWITCH_Q1**: Instanz des Typs adapter::events::unidirectional::AX_E_SWITCH.
- **E_SWITCH_Q2**: Instanz des Typs adapter::events::unidirectional::AX_E_SWITCH.
- **E_SWITCH_Q3**: Instanz des Typs adapter::events::unidirectional::AX_E_SWITCH.
- **E_SWITCH_Q4**: Instanz des Typs adapter::events::unidirectional::AX_E_SWITCH.
- **SPLIT_1**: Instanz des Typs adapter::events::unidirectional::AX_SPLIT_2.
- **SPLIT_2**: Instanz des Typs adapter::events::unidirectional::AX_SPLIT_2.
- **SPLIT_3**: Instanz des Typs adapter::events::unidirectional::AX_SPLIT_2.
- **SPLIT_4**: Instanz des Typs adapter::events::unidirectional::AX_SPLIT_2.

Verbindungen und Schnittstellen

Adapterverbindungen:

- E_SR_Ausfahren_Cyl_1.Q -> SPLIT_1.IN
- SPLIT_1.OUT1 -> DigitalOutput_Q1.OUT
- SPLIT_1.OUT2 -> E_SWITCH_Q1.G
- E_SR_Ausfahren_Cyl_2.Q -> SPLIT_2.IN
- SPLIT_2.OUT1 -> DigitalOutput_Q2.OUT

- SPLIT_2.OUT2 -> E_SWITCH_Q2.G
- E_SR_Einfahren_Cyl_2.Q -> SPLIT_3.IN
- SPLIT_3.OUT1 -> DigitalOutput_Q3.OUT
- SPLIT_3.OUT2 -> E_SWITCH_Q3.G
- E_SR_Einfahren_Cyl_1.Q -> SPLIT_4.IN
- SPLIT_4.OUT1 -> DigitalOutput_Q4.OUT
- SPLIT_4.OUT2 -> E_SWITCH_Q4.G

Ereignisverbindungen:

- SoftKey_UP_F1.IND -> E_SR_Ausfahren_Cyl_1.S
- E_DELAY.EO -> E_SR_Einfahren_Cyl_2.S
- E_REND_Ausfahren_Cyl_1.EO -> E_SR_Ausfahren_Cyl_1.R
- E_REND_Ausfahren_Cyl_1.EO -> E_SR_Ausfahren_Cyl_2.S
- SoftKey_F2_DOWN.IND -> E_REND_Ausfahren_Cyl_1.EI2
- SoftKey_F3_DOWN.IND -> E_REND_Ausfahren_Cyl_2.EI2
- E_REND_Ausfahren_Cyl_2.EO -> E_SR_Ausfahren_Cyl_2.R
- E_REND_Einfahren_Cyl_1.EO -> E_SR_Einfahren_Cyl_1.R
- SoftKey_F9_DOWN.IND -> E_REND_Einfahren_Cyl_1.EI2
- E_REND_Einfahren_Cyl_2.EO -> E_SR_Einfahren_Cyl_1.S
- E_REND_Einfahren_Cyl_2.EO -> E_SR_Einfahren_Cyl_2.R
- SoftKey_F8_DOWN.IND -> E_REND_Einfahren_Cyl_2.EI2
- E_REND_Ausfahren_Cyl_2.EO -> E_DELAY.START
- SoftKey_UP_F1.IND -> E_REND_Ausfahren_Cyl_1.R
- SoftKey_F2_DOWN.IND -> E_REND_Ausfahren_Cyl_2.R
- E_DELAY.EO -> E_REND_Einfahren_Cyl_2.R
- SoftKey_F8_DOWN.IND -> E_REND_Einfahren_Cyl_1.R
- E_SWITCH_Q1.EO1 -> E_REND_Ausfahren_Cyl_1.EI1
- E_SWITCH_Q2.EO1 -> E_REND_Ausfahren_Cyl_2.EI1
- E_SWITCH_Q3.EO1 -> E_REND_Einfahren_Cyl_2.EI1
- E_SWITCH_Q4.EO1 -> E_REND_Einfahren_Cyl_1.EI1

Hinweise aus dem Modell

START-Knopf Ausfahren Endlage Ausfahren_Cyl_1 Endlage Ausfahren_Cyl_2 Ausfahren_Cyl_1 Ausfahren_Cyl_2 Einfahren Zeit gesteuert Endlage Einfahren_Cyl_1 Einfahren_Cyl_2 Endlage Einfahren_Cyl_2 Einfahren_Cyl_1

Programmablauf und Funktionsweise

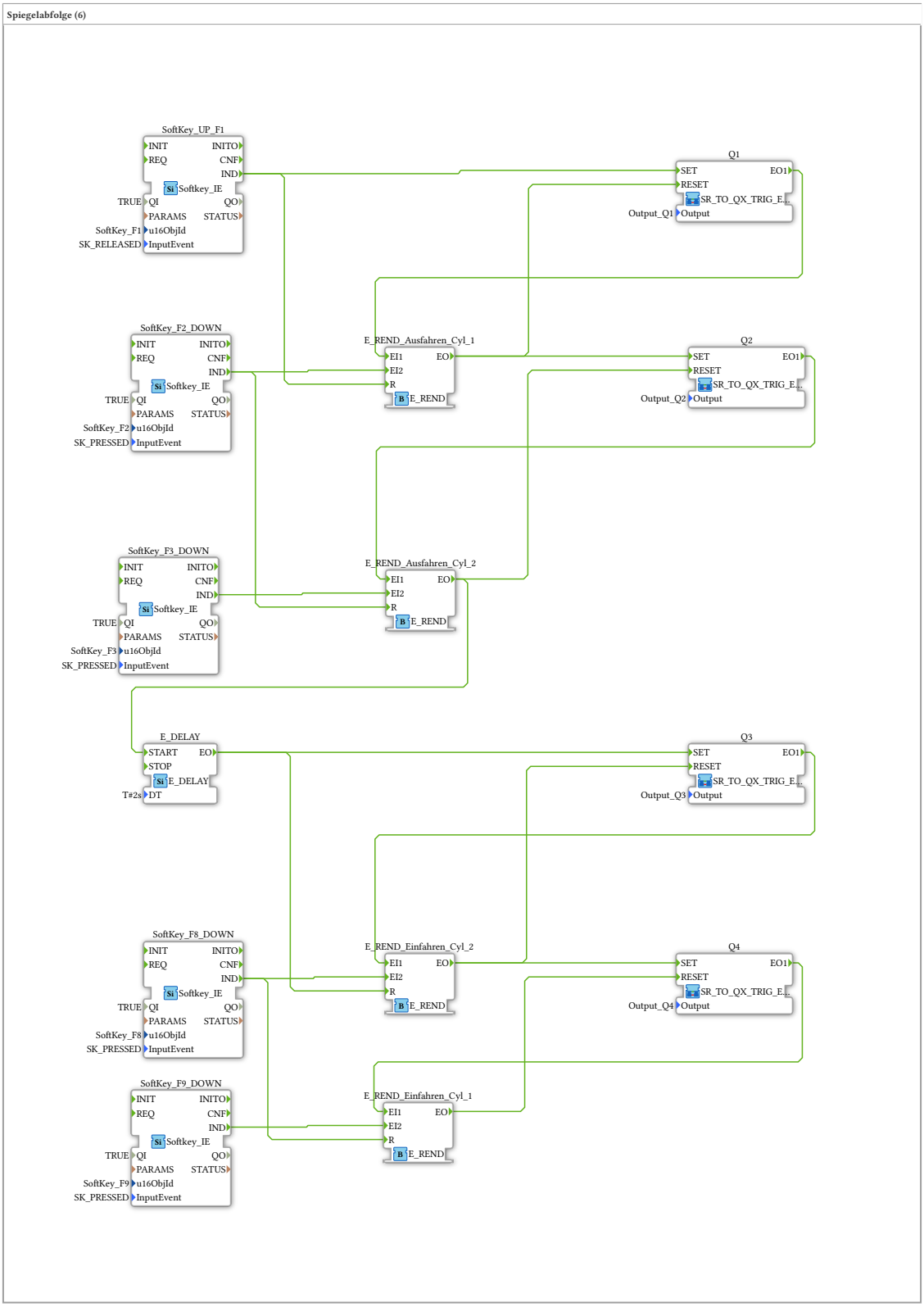
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_025_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.261 Uebung_026_AX: Spiegelabfolge (6)

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_026_AX (Spiegelabfolge (6)).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Spiegelabfolge (6)**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_026_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **SoftKey_UP_F1**: Instanz des Typs isobus::UT::io::Softkey::Softkey_IE.
  - Parameter QI = TRUE
  - Parameter u16ObjId = SoftKey_F1
  - Parameter InputEvent = SK_RELEASED
- **SoftKey_F2_DOWN**: Instanz des Typs isobus::UT::io::Softkey::Softkey_IE.
  - Parameter QI = TRUE
  - Parameter u16ObjId = SoftKey_F2
  - Parameter InputEvent = SK_PRESSED
- **SoftKey_F3_DOWN**: Instanz des Typs isobus::UT::io::Softkey::Softkey_IE.
  - Parameter QI = TRUE
  - Parameter u16ObjId = SoftKey_F3
  - Parameter InputEvent = SK_PRESSED
- **SoftKey_F9_DOWN**: Instanz des Typs isobus::UT::io::Softkey::Softkey_IE.
  - Parameter QI = TRUE
  - Parameter u16ObjId = SoftKey_F9
  - Parameter InputEvent = SK_PRESSED
- **SoftKey_F8_DOWN**: Instanz des Typs isobus::UT::io::Softkey::Softkey_IE.
  - Parameter QI = TRUE
  - Parameter u16ObjId = SoftKey_F8
  - Parameter InputEvent = SK_PRESSED
- **E_DELAY**: Instanz des Typs iec61499::events::E_DELAY.
  - Parameter DT = T#2s
- **E_REND_Ausfahren_Cyl_1**: Instanz des Typs iec61499::events::E_REND.
- **E_REND_Ausfahren_Cyl_2**: Instanz des Typs iec61499::events::E_REND.
- **E_REND_Einfahren_Cyl_2**: Instanz des Typs iec61499::events::E_REND.
- **E_REND_Einfahren_Cyl_1**: Instanz des Typs iec61499::events::E_REND.

Verbindungen und Schnittstellen

Ereignisverbindungen:

- SoftKey_F2_DOWN.IND -> E_REND_Ausfahren_Cyl_1.EI2
- SoftKey_F3_DOWN.IND -> E_REND_Ausfahren_Cyl_2.EI2
- SoftKey_F9_DOWN.IND -> E_REND_Einfahren_Cyl_1.EI2
- SoftKey_F8_DOWN.IND -> E_REND_Einfahren_Cyl_2.EI2
- E_REND_Ausfahren_Cyl_2.EO -> E_DELAY.START
- SoftKey_UP_F1.IND -> E_REND_Ausfahren_Cyl_1.R
- SoftKey_F2_DOWN.IND -> E_REND_Ausfahren_Cyl_2.R
- E_DELAY.EO -> E_REND_Einfahren_Cyl_2.R
- SoftKey_F8_DOWN.IND -> E_REND_Einfahren_Cyl_1.R
- E_REND_Einfahren_Cyl_2.EO -> Q4.SET
- E_REND_Einfahren_Cyl_1.EO -> Q4.RESET
- Q4.EO1 -> E_REND_Einfahren_Cyl_1.EI1
- E_REND_Ausfahren_Cyl_1.EO -> Q2.SET
- E_REND_Ausfahren_Cyl_2.EO -> Q2.RESET
- Q2.EO1 -> E_REND_Ausfahren_Cyl_2.EI1
- E_DELAY.EO -> Q3.SET
- E_REND_Einfahren_Cyl_2.EO -> Q3.RESET
- Q3.EO1 -> E_REND_Einfahren_Cyl_2.EI1
- SoftKey_UP_F1.IND -> Q1.SET
- E_REND_Ausfahren_Cyl_1.EO -> Q1.RESET
- Q1.EO1 -> E_REND_Ausfahren_Cyl_1.EI1

Hinweise aus dem Modell

START-Knopf Ausfahren Endlage Ausfahren_Cyl_1 Endlage Ausfahren_Cyl_2 Ausfahren_Cyl_1 Ausfahren_Cyl_2 Einfahren Zeit gesteuert Endlage Einfahren_Cyl_1 Einfahren_Cyl_2 Endlage Einfahren_Cyl_2 Einfahren_Cyl_1

Programmablauf und Funktionsweise

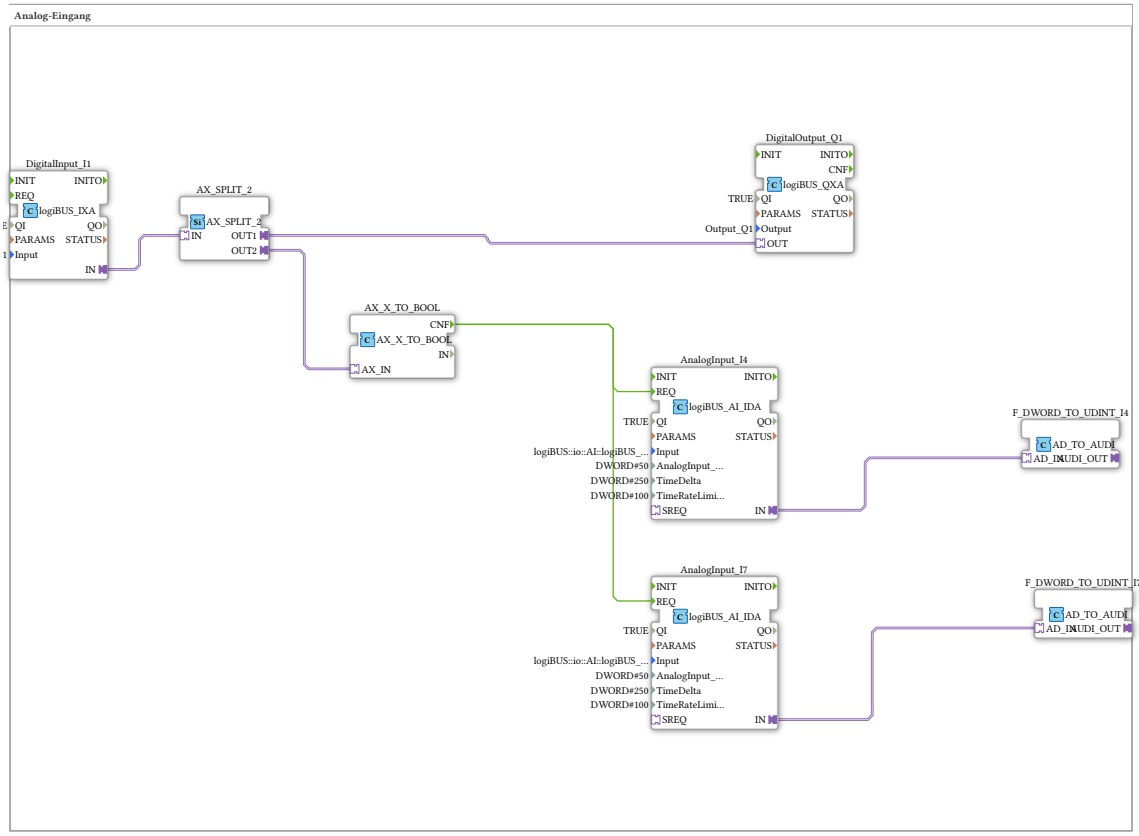
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_026_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.262 Uebung_028_AX: Analog-Eingang

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_028_AX (Analog-Eingang).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Analog-Eingang**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_028_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter PARAMS =
  - Parameter Output = Output_Q1
- **DigitalInput_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter PARAMS =
  - Parameter Input = Input_I1

- **AnalogInput_I4**: Instanz des Typs logiBUS::io::AI::logiBUS_AI_IDA.
  - Parameter QI = TRUE
  - Parameter Input = logiBUS::io::AI::logiBUS_AI::AnalogInput_I4
  - Parameter AnalogInput_hysteresis = 50
  - Parameter TimeDelta = 250
  - Parameter TimeRateLimit = 100
- **AnalogInput_I7**: Instanz des Typs logiBUS::io::AI::logiBUS_AI_IDA.
  - Parameter QI = TRUE
  - Parameter Input = logiBUS::io::AI::logiBUS_AI::AnalogInput_I7
  - Parameter AnalogInput_hysteresis = 50
  - Parameter TimeDelta = 250
  - Parameter TimeRateLimit = 100
- **F_DWORD_TO_UDINT_I7**: Instanz des Typs adapter::conversion::unidirectional::AD_TO_AUDI.
- **F_DWORD_TO_UDINT_I4**: Instanz des Typs adapter::conversion::unidirectional::AD_TO_AUDI.
- **AX_X_TO_BOOL**: Instanz des Typs adapter::conversion::unidirectional::AX_X_TO_BOOL.
- **AX_SPLIT_2**: Instanz des Typs adapter::events::unidirectional::AX_SPLIT_2.

Verbindungen und Schnittstellen

Adapterverbindungen:

- AnalogInput_I7.IN -> F_DWORD_TO_UDINT_I7.AD_IN
- AX_SPLIT_2.OUT1 -> DigitalOutput_Q1.OUT
- AnalogInput_I4.IN -> F_DWORD_TO_UDINT_I4.AD_IN
- DigitalInput_I1.IN -> AX_SPLIT_2.IN
- AX_SPLIT_2.OUT2 -> AX_X_TO_BOOL.AX_IN

Ereignisverbindungen:

- AX_X_TO_BOOL.CNF -> AnalogInput_I4.REQ
- AX_X_TO_BOOL.CNF -> AnalogInput_I7.REQ

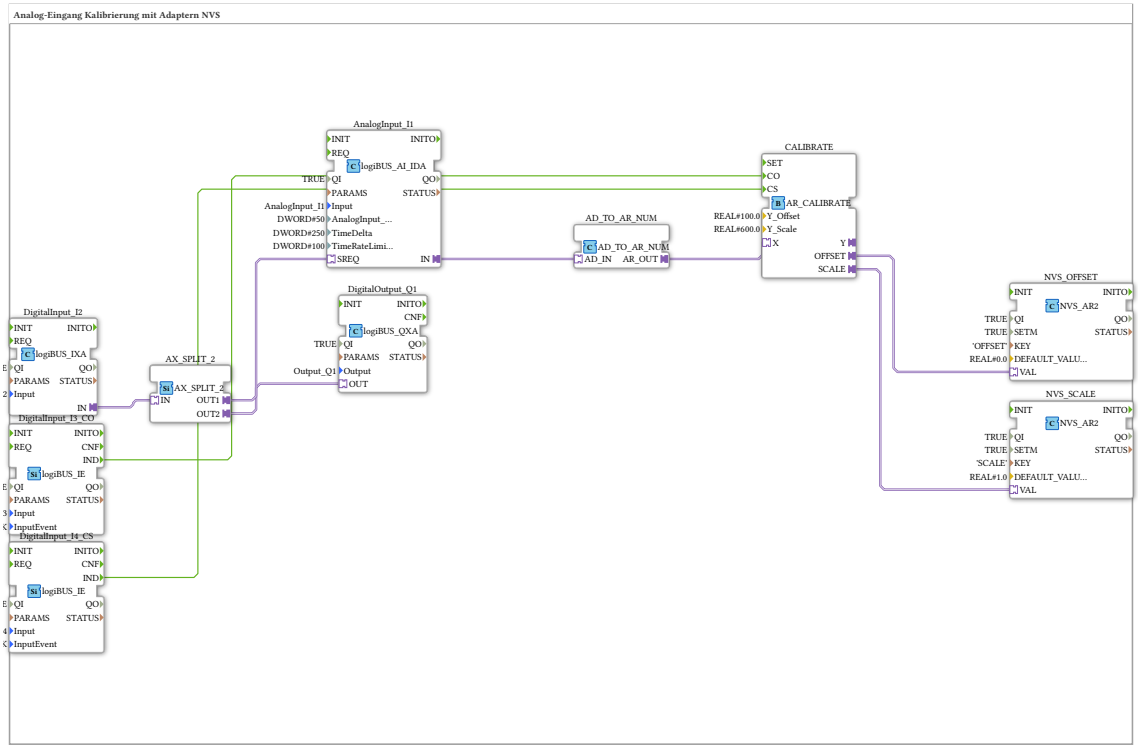
Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_028_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.263 Uebung_028a2_AR: Analog-Eingang Kalibrierung mit Adaptern NVS



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung realisiert eine Kalibrierung für einen analogen Eingang. Ein Analogwert wird eingelesen, über eine Adapterkonvertierung an einen Kalibrierungsbaustein übergeben und die ermittelten Offset- und Skalierungswerte werden dauerhaft im NVS-Speicher (Non-Volatile Storage) abgelegt. Zwei digitale Eingänge steuern den Kalibriermodus (Offset und Scale). Ein weiterer digitaler Eingang dient als Trigger für die analoge Abtastung und wird gleichzeitig auf einen digitalen Ausgang durchgeschleift.

Verwendete Funktionsbausteine (FBs)

Name	Typ	Parameter
DigitalInput_I2	`logiBUS::io::DI::logiBUS_IXA`	QI = TRUE, Input = Input_I2
DigitalOutput_Q1	`logiBUS::io::DQ::logiBUS_QXA`	QI = TRUE, Output = Output_Q1
AnalogInput_I1	`logiBUS::io::AI::logiBUS_AI_IDA`	QI = TRUE, Input = AnalogInput_I1, AnalogInput_hysteresis = 50, TimeDelta = 250, TimeRateLimit = 100
CALIBRATE	`adapter::Engineering::measurements::AR_CALIBRATE`	X = Offset = 100.0, Y_Scale = 600.0
NVS_OFFSET	`logiBUS::storage::esp32_nvs::NVS_AR2`	QI = TRUE, SETM = TRUE, KEY = ‚OFFSET‘, DEFAULT_VALUE = 0.0
NVS_SCALE	`logiBUS::storage::esp32_nvs::NVS_AR2`	QI = TRUE, SETM = TRUE, KEY = ‚SCALE‘, DEFAULT_VALUE = 1.0
DigitalInput_I3_CO	`logiBUS::io::DI::logiBUS_IE`	QI = TRUE, Input = Input_I3, InputEvent = BUTTON_SINGLE_CLICK
DigitalInput_I4_CS	`logiBUS::io::DI::logiBUS_IE`	QI = TRUE, Input = Input_I4, InputEvent = BUTTON_SINGLE_CLICK
AX_SPLIT_2	`adapter::events::unidirectional::AX_SPLIT_2`	(keine Parameter)

AD_TO_AR_NUM	logiBUS::conversion::unidirectional::AD_TO_AR_NUM	(keine Parameter)

Kurzbeschreibung der Bausteine

- **DigitalInput_I2**: Liest einen digitalen Eingang (Input_I2) und leitet das Ereignis über den Adapterausgang `IN` weiter.
- **DigitalOutput_Q1**: Gibt ein digitales Signal auf den Ausgang `OUT` (Output_Q1) aus, wenn der Adaptereingang `OUT` angesteuert wird.
- **AnalogInput_I1**: Analoger Eingangsbaustein. Bei einer Anfrage (SREQ) liefert er einen Analogwert. Parametriert mit Hysterese, Zeitdelta und Ratenbegrenzung.
- **CALIBRATE**: Führt die Kalibrierung durch. Empfängt den Rohwert am Adapter `X` und die Steuersignale `CO` (Calibrate Offset) und `CS` (Calibrate Scale). Gibt den berechneten Offset und die Skalierung über die Adapter `OFFSET` und `SCALE` aus.
- **NVS_OFFSET / NVS_SCALE**: Speichern einen Gleitkommawert im nichtflüchtigen Speicher unter den Keys ‚OFFSET' bzw. ‚SCALE' mit vorgegebenen Defaultwerten.
- **DigitalInput_I3_CO / DigitalInput_I4_CS**: Zusätzliche digitale Eingänge für die Kalibrierungssteuerung (Input_I3 = Offset-Kalibrierung, Input_I4 = Scale-Kalibrierung).
- **AX_SPLIT_2**: Verteilt ein eingehendes Adapterereignis auf zwei Ausgänge (OUT1 und OUT2).
- **AD_TO_AR_NUM**: Instanz des gleichnamigen Typs `AD_TO_AR_NUM`. Wandelt den Analogdaten-Adapter (`AD_IN`) numerisch (AD → UDINT → REAL) in einen Real-Adapter (`AR_OUT`) um. Ein einfaches `AD_TO_AR` wäre dagegen wie ein „reinterpret_cast" (Bitmuster).

Programmablauf und Verbindungen

Der Ablauf wird durch den digitalen Eingang `Input_I2` gestartet:

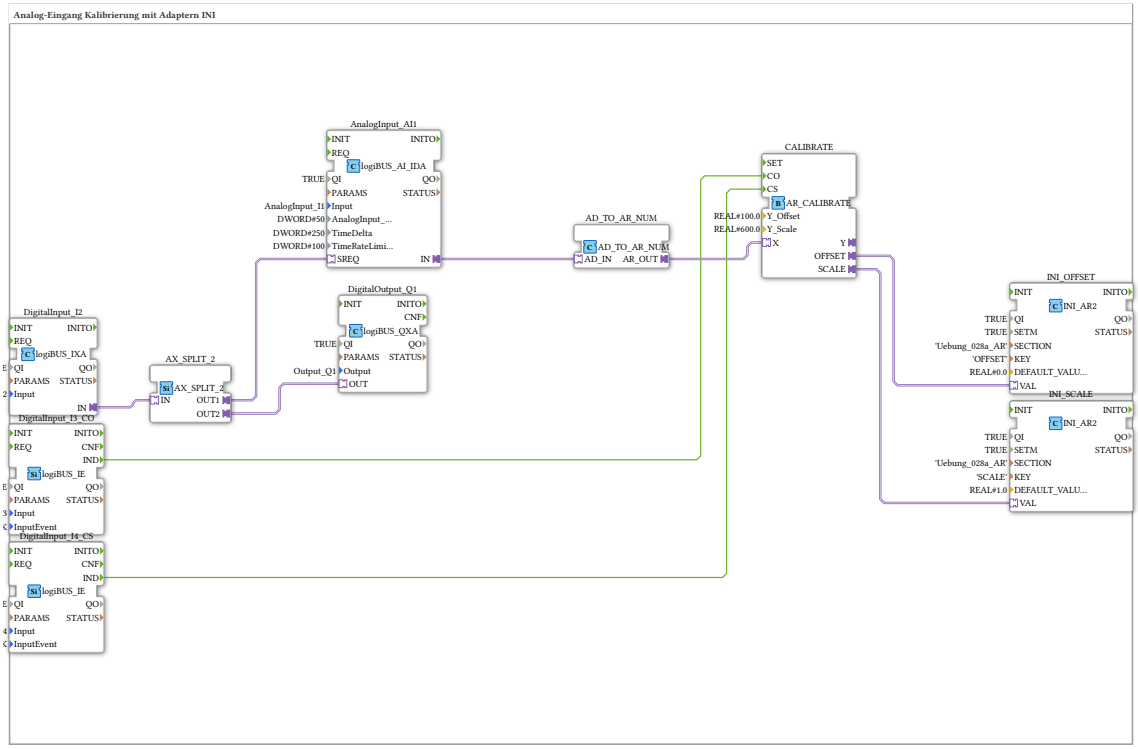
1. **Ereignisverteilung**: Das von `DigitalInput_I2` kommende Ereignis (Adapter `IN`) gelangt zu `AX_SPLIT_2`. Dieses teilt das Ereignis auf:
   - **OUT1** → verbunden mit `AnalogInput_I1.SREQ` → löst die analoge Abtastung aus.
   - **OUT2** → verbunden mit `DigitalOutput_Q1.OUT` → der digitale Ausgang `Output_Q1` wird gesetzt.
2. **Analoger Messwert**: `AnalogInput_I1` liefert nach der Abtastung einen Analogdaten-Adapter über seinen Ausgang `IN`. Dieser wird an `AD_TO_AR_NUM.AD_IN` übergeben.
3. **Konvertierung**:
   - `AD_TO_AR_NUM` wandelt den Analogdaten-Adapter (`AD_IN`) numerisch in einen Real-Adapter (`AR_OUT`) um.
   - Der Real-Adapter wird an den Kalibrierungseingang `CALIBRATE.X` weitergeleitet.
4. **Kalibrierung**: Gleichzeitig liegen an `CALIBRATE.CO` der Eingang `Input_I3` (über `DigitalInput_I3_CO`) und an `CALIBRATE.CS` der Eingang `Input_I4` (über `DigitalInput_I4_CS`) an. Je nach aktiviertem Steuersignal berechnet `CALIBRATE` den neuen Offset oder die neue Skalierung. Die Voreinstellungen (Y_Offset = 100.0, Y_Scale = 600.0) dienen als Basis.
5. **Persistente Speicherung**:
   - Der ermittelte Offset (Adapter `OFFSET`) wird an `NVS_OFFSET.VAL` übergeben und unter dem Key ‚OFFSET' abgelegt.
   - Der ermittelte Skalierungsfaktor (Adapter `SCALE`) wird an `NVS_SCALE.VAL` übergeben und unter dem Key ‚SCALE' abgelegt.

Die Kalibrierungswerte bleiben damit auch nach einem Neustart der Steuerung erhalten.

Zusammenfassung

Die Übung demonstriert eine vollständige Signalkette von der digitalen Triggerung über die analoge Erfassung, Adapterkonvertierung, Kalibrierung bis zur dauerhaften Speicherung der Korrekturwerte im NVS. Sie verdeutlicht den Umgang mit mehreren Adaptertypen (Ereignis-, Daten- und Real-Adapter) und das Zusammenspiel von Standard-FBs der logiBUS-Bibliothek mit speziellen Engineering-Bausteinen. Die Adapterkonvertierung zwischen Analogdaten und Real-Werten erfolgt über den dafür vorgesehenen Baustein `AD_TO_AR_NUM` statt über ein „reinterpret_cast".

2.1.2.264 Uebung_028a_AR: Analog-Eingang Kalibrierung mit Adaptern INI



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert die Kalibrierung eines analogen Eingangssignals mithilfe von Adaptern und der Speicherung der Kalibrierungsparameter (Offset und Skalierung) in einer INI-Datei. Der Funktionsbaustein `AR_CALIBRATE` führt die Zwei-Punkt-Kalibrierung durch. Die Kalibrierung wird über zwei digitale Eingänge (`Input_I3`, `Input_I4`) ausgelöst, die Ergebnisse werden in zwei separaten Speicherbausteinen (`INI_AR2`) abgelegt. Die Konvertierung zwischen den Adaptertypen `AD` und `AR` erfolgt numerisch über einen einzelnen Adapterkonverter (`AD_TO_AR_NUM`).

Verwendete Funktionsbausteine (FBs)

Die Übung verwendet ausschließlich direkt instanziierte Funktionsbausteine (keine Unter-Applikationen).

- **DigitalInput_I2**: logiBUS Digitaleingang
  - **Typ**: `logiBUS::io::DI::logiBUS_IXA`
  - **Parameter**: `QI = TRUE`, `Input = Input_I2`
  - **Erklärung**: Liest den Zustand des Eingangs `I2` und gibt ihn über den Adapter `IN` weiter. Dient als Trigger für die Messanforderung des Analogeingangs.
- **DigitalOutput_Q1**: logiBUS Digitalausgang
  - **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
  - **Parameter**: `QI = TRUE`, `Output = Output_Q1`
  - **Erklärung**: Schaltet den Ausgang `Q1` entsprechend dem Zustand von `I2` (über `AX_SPLIT_2.OUT2`).
- **AnalogInput_AI1**: logiBUS Analogeingang
  - **Typ**: `logiBUS::io::AI::logiBUS_AI_IDA`
  - **Parameter**: `QI = TRUE`, `Input = AnalogInput_I1`, `AnalogInput_hysteresis = 50`, `TimeDelta = 250`, `TimeRateLimit = 100`
  - **Erklärung**: Liest den analogen Eingang `I1` und stellt den Wert über den Adapter `IN` bereit. Über den Adapter `SREQ` kann eine Messung angefordert werden. Die Parameter dienen der Filterung (Hysterese, Zeitdelta, Ratenbegrenzung).
- **CALIBRATE**: Zwei-Punkt-Kalibrierung
  - **Typ**: `adapter::Engineering::measurements::AR_CALIBRATE`
  - **Parameter**: `Y_Offset = 100.0`, `Y_Scale = 600.0`

  - **Erklärung**: Berechnet den kalibrierten Wert nach der Formel `Y = (X + OFFSET) * SCALE`. Mit dem Event `CO` (Calibrate Offset) wird `OFFSET := Y_Offset - X` gesetzt, mit dem Event `CS` (Calibrate Scale) wird `SCALE := Y_Scale / (X + OFFSET)` gesetzt. `Y_Offset` und `Y_Scale` sind dabei Sollwerte auf der Ausgangsseite. `CO` muss vor `CS` ausgeführt werden, da `CS` den Offset verwendet. Die berechneten Werte werden über die bidirektionalen Adapter `OFFSET` und `SCALE` ausgegeben bzw. eingelesen. Der kalibrierte Ausgang `Y` ist in dieser Übung nicht verschaltet.
- **INI_OFFSET**: Speicherbaustein für den Offset
  - **Typ**: `eclipse4diac::storage::INI_AR2`
  - **Parameter**: `QI = TRUE`, `SETM = TRUE`, `SECTION = 'Uebung_028a_AR'`, `KEY = 'OFFSET'`, `DEFAULT_VALUE = 0.0`
  - **Erklärung**: Lädt den Wert aus der INI-Datei (Abschnitt `Uebung_028a_AR`, Schlüssel `OFFSET`; ohne Eintrag gilt `DEFAULT_VALUE`) und liefert ihn über den Adapter `VAL`; neue Werte von `VAL` werden gespeichert. `SETM = TRUE` aktiviert die Spiegelung beim Setzen.
- **INI_SCALE**: Speicherbaustein für die Skalierung
  - **Typ**: `eclipse4diac::storage::INI_AR2`
  - **Parameter**: `QI = TRUE`, `SETM = TRUE`, `SECTION = 'Uebung_028a_AR'`, `KEY = 'SCALE'`, `DEFAULT_VALUE = 1.0`
  - **Erklärung**: Analog zu `INI_OFFSET`, jedoch für den Skalierungsfaktor (Schlüssel `SCALE`).
- **DigitalInput_I3_CO**: logiBUS Eingang mit Klick-Erkennung für die Offset-Kalibrierung
  - **Typ**: `logiBUS::io::DI::logiBUS_IE`
  - **Parameter**: `QI = TRUE`, `Input = Input_I3`, `InputEvent = BUTTON_SINGLE_CLICK`
  - **Erklärung**: Ein Einzelklick auf `I3` löst `CALIBRATE.CO` aus.
- **DigitalInput_I4_CS**: logiBUS Eingang mit Klick-Erkennung für die Skalen-Kalibrierung
  - **Typ**: `logiBUS::io::DI::logiBUS_IE`
  - **Parameter**: `QI = TRUE`, `Input = Input_I4`, `InputEvent = BUTTON_SINGLE_CLICK`
  - **Erklärung**: Ein Einzelklick auf `I4` löst `CALIBRATE.CS` aus.
- **AX_SPLIT_2**: Adapter-Splitter
  - **Typ**: `adapter::events::unidirectional::AX_SPLIT_2`
  - **Parameter**: keine
  - **Erklärung**: Verteilt das Signal von `DigitalInput_I2` gleichzeitig auf die Messanforderung `AnalogInput_AI1.SREQ` (`OUT1`) und auf `DigitalOutput_Q1` (`OUT2`).
- **AD_TO_AR_NUM**: Adapterkonverter (numerisch)
  - **Typ**: `adapter::conversion::unidirectional::AD_TO_AR_NUM`
  - **Parameter**: keine
  - **Erklärung**: Wandelt den vom Analogeingang gelieferten `AD`-Adapter numerisch (AD → UDINT → REAL) in den `AR`-Adapter um, den `AR_CALIBRATE` am Eingang `X` erwartet.

Programmablauf und Verbindungen

1. `DigitalInput_I2.IN` → `AX_SPLIT_2.IN` (AdapterConnection): das Signal von `I2` wird in den Splitter geführt.
2. `AX_SPLIT_2.OUT1` → `AnalogInput_AI1.SREQ` und `AX_SPLIT_2.OUT2` → `DigitalOutput_Q1.OUT` (AdapterConnections): `Q1` folgt `I2`, gleichzeitig wird eine Messung am Analogeingang angefordert.
3. `AnalogInput_AI1.IN` → `AD_TO_AR_NUM.AD_IN` und `AD_TO_AR_NUM.AR_OUT` → `CALIBRATE.X` (AdapterConnections): der Messwert wird über den numerischen Konverter an den Kalibrierbaustein geführt.
4. `DigitalInput_I3_CO.IND` → `CALIBRATE.CO` (EventConnection): ein Klick auf `I3` löst die Offset-Kalibrierung aus.
5. `DigitalInput_I4_CS.IND` → `CALIBRATE.CS` (EventConnection): ein Klick auf `I4` löst die Skalen-Kalibrierung aus.
6. `CALIBRATE.OFFSET` → `INI_OFFSET.VAL` und `CALIBRATE.SCALE` → `INI_SCALE.VAL` (AdapterConnections): Offset und Skalierung werden in der INI-Datei abgelegt bzw. beim Start von dort geladen, sodass die Kalibrierung dauerhaft erhalten bleibt.

Wichtige Anmerkung (Kommentar im Modell): Ein `AD_TO_AR` wäre wie ein „reinterpret_cast“ (Bitmuster). `AD_TO_AR_NUM` wandelt dagegen den Zahlenwert (AD → UDINT → REAL).

Lernziele dieser Übung:

- Umgang mit analogen Eingangsadaptern und deren Parametrierung.
- Einsatz des numerischen Adapterkonverters `AD_TO_AR_NUM` zur Typanpassung.
- Verwendung von INI-Speicherbausteinen zum dauerhaften Ablegen von Konfigurationsparametern.
- Verständnis der Zwei-Punkt-Kalibrierung (Offset, dann Skalierung).

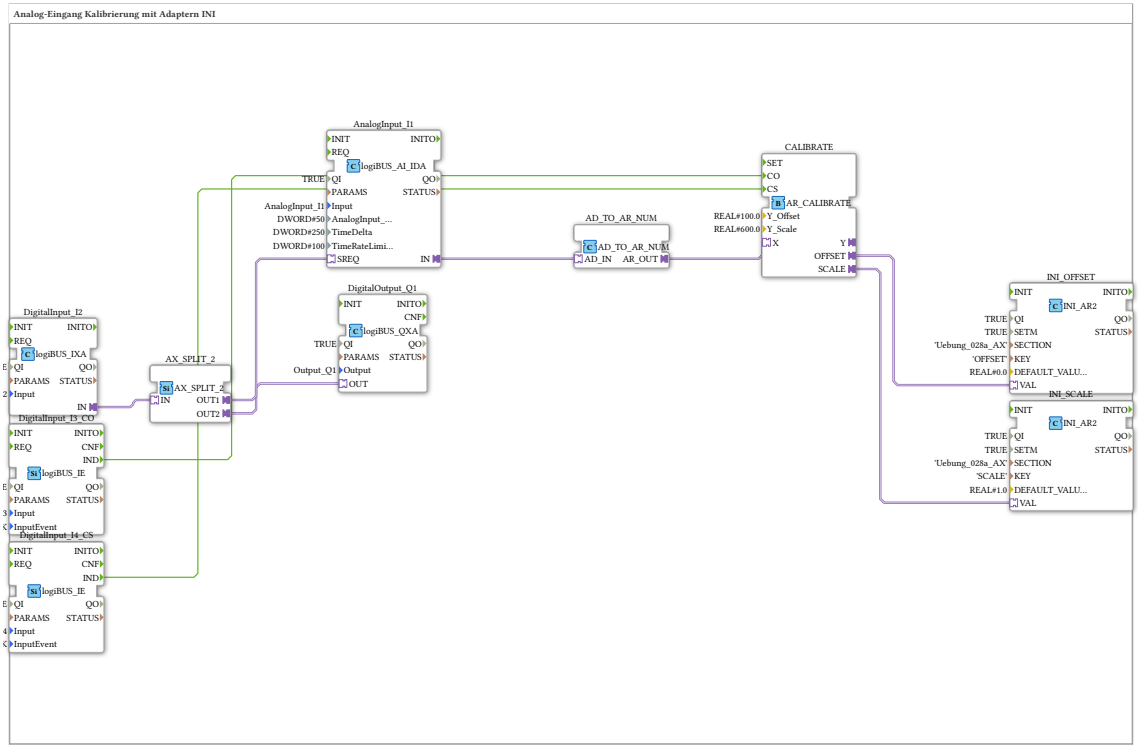
Zusammenfassung

Die Übung `Uebung_028a_AR` implementiert eine Analog-Eingangs-Kalibrierung, bei der Offset und Skalierung über zwei Klick-Eingänge (`I3`, `I4`) eingelernt und in einer INI-Datei persistiert werden. Eine Messung wird über `I2` angefordert, das gleichzeitig `Q1` schaltet. Der Adapterkonverter `AD_TO_AR_NUM` demonstriert die wertrichtige (numerische) Weiterverarbeitung analoger Signale in der 4diac IDE.

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

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

2.1.2.265 Uebung_028a_AX: Analog-Eingang Kalibrierung mit Adaptern INI



• — ○ • — ○ • — ○ •

Einleitung

Diese Übung kalibriert den Analog-Eingang `I1` mit dem Baustein `AR_CALIBRATE`. Der Analogwert wird als Adapter geführt und über den Baustein `AD_TO_AR_NUM` numerisch (AD ⟶ UDINT ⟶ REAL) in einen `REAL`-Wert gewandelt. Die berechneten Kalibrierwerte `OFFSET` und `SCALE` werden über zwei `INI_AR2`-Bausteine in einer INI-Datei gespeichert.

> **Wichtig**: Ein `AD_TO_AR` wäre wie ein „reinterpret_cast“ (Bitmuster). `AD_TO_AR_NUM` wandelt dagegen den Zahlenwert (AD ⟶ UDINT ⟶ REAL).

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_I2**: logiBUS Digitaleingang
  - **Typ**: `logiBUS::io::DI::logiBUS_IXA`
  - **Parameter**: `QI = TRUE`, `Input = Input_I2`
  - **Erklärung**: Liefert den Zustand von `I2` als Adapter.
- **AX_SPLIT_2**: Verteilung des Adapters auf zwei Ziele
  - **Typ**: `adapter::events::unidirectional::AX_SPLIT_2`
  - **Parameter**: keine
  - **Erklärung**: `OUT1` geht an `AnalogInput_I1.SREQ`, `OUT2` an den Ausgang `Q1`.
- **DigitalOutput_Q1**: logiBUS Digitalausgang

  - **Typ**: logiBUS::io::DQ::logiBUS_QXA
  - **Parameter**: QI = TRUE, Output = Output_Q1
  - **Erklärung**: Gibt den Wert von AX_SPLIT_2.OUT2 am physischen Ausgang Q1 aus.
- **AnalogInput_I1**: logiBUS Analogeingang
  - **Typ**: logiBUS::io::AI::logiBUS_AI_IDA
  - **Parameter**: QI = TRUE, Input = AnalogInput_I1, AnalogInput_hysteresis = 50, TimeDelta = 250, TimeRateLimit = 100
  - **Erklärung**: Liefert den Analogwert von I1 über den Adapter IN.
- **AD_TO_AR_NUM**: numerische Konvertierung
  - **Typ**: adapter::conversion::unidirectional::AD_TO_AR_NUM
  - **Parameter**: keine
  - **Erklärung**: Wandelt AD_IN numerisch (AD → UDINT → REAL) in AR_OUT.
- **DigitalInput_I3_CO**: Taster für das Ereignis CO
  - **Typ**: logiBUS::io::DI::logiBUS_IE
  - **Parameter**: QI = TRUE, Input = Input_I3, InputEvent = BUTTON_SINGLE_CLICK
  - **Erklärung**: Meldet einen Einzelklick auf I3 als IND.
- **DigitalInput_I4_CS**: Taster für das Ereignis CS
  - **Typ**: logiBUS::io::DI::logiBUS_IE
  - **Parameter**: QI = TRUE, Input = Input_I4, InputEvent = BUTTON_SINGLE_CLICK
  - **Erklärung**: Meldet einen Einzelklick auf I4 als IND.
- **CALIBRATE**: Kalibrierung
  - **Typ**: adapter::Engineering::measurements::AR_CALIBRATE
  - **Parameter**: Y_Offset = 100.0, Y_Scale = 600.0
  - **Erklärung**: Erhält den gewandelten Messwert über X und die Ereignisse CO und CS; liefert OFFSET und SCALE als Adapter.
- **INI_OFFSET**: Ablage von OFFSET in der INI-Datei
  - **Typ**: eclipse4diac::storage::INI_AR2
  - **Parameter**: QI = TRUE, SETM = TRUE, SECTION = 'Uebung_028a_AX', KEY = 'OFFSET', DEFAULT_VALUE = 0.0
  - **Erklärung**: Speichert den Wert von CALIBRATE.OFFSET (Eingang VAL) unter dem Schlüssel OFFSET.
- **INI_SCALE**: Ablage von SCALE in der INI-Datei
  - **Typ**: eclipse4diac::storage::INI_AR2
  - **Parameter**: QI = TRUE, SETM = TRUE, SECTION = 'Uebung_028a_AX', KEY = 'SCALE', DEFAULT_VALUE = 1.0
  - **Erklärung**: Speichert den Wert von CALIBRATE.SCALE (Eingang VAL) unter dem Schlüssel SCALE.

Programmablauf und Verbindungen

1. DigitalInput_I2.IN → AX_SPLIT_2.IN; AX_SPLIT_2.OUT1 → AnalogInput_I1.SREQ und AX_SPLIT_2.OUT2 → DigitalOutput_Q1.OUT (AdapterConnections): I2 schaltet Q1 und ist gleichzeitig mit SREQ des Analogeingangs verbunden.
2. AnalogInput_I1.IN → AD_TO_AR_NUM.AD_IN und AD_TO_AR_NUM.AR_OUT → CALIBRATE.X (AdapterConnections): Der Analogwert wird numerisch konvertiert und als Messwert X an CALIBRATE gegeben.
3. DigitalInput_I3_CO.IND → CALIBRATE.CO und DigitalInput_I4_CS.IND → CALIBRATE.CS (EventConnections): Die beiden Taster I3 und I4 lösen die Ereignisse CO und CS aus.
4. CALIBRATE.OFFSET → INI_OFFSET.VAL und CALIBRATE.SCALE → INI_SCALE.VAL (AdapterConnections): Die Kalibrierwerte werden in der INI-Datei abgelegt.

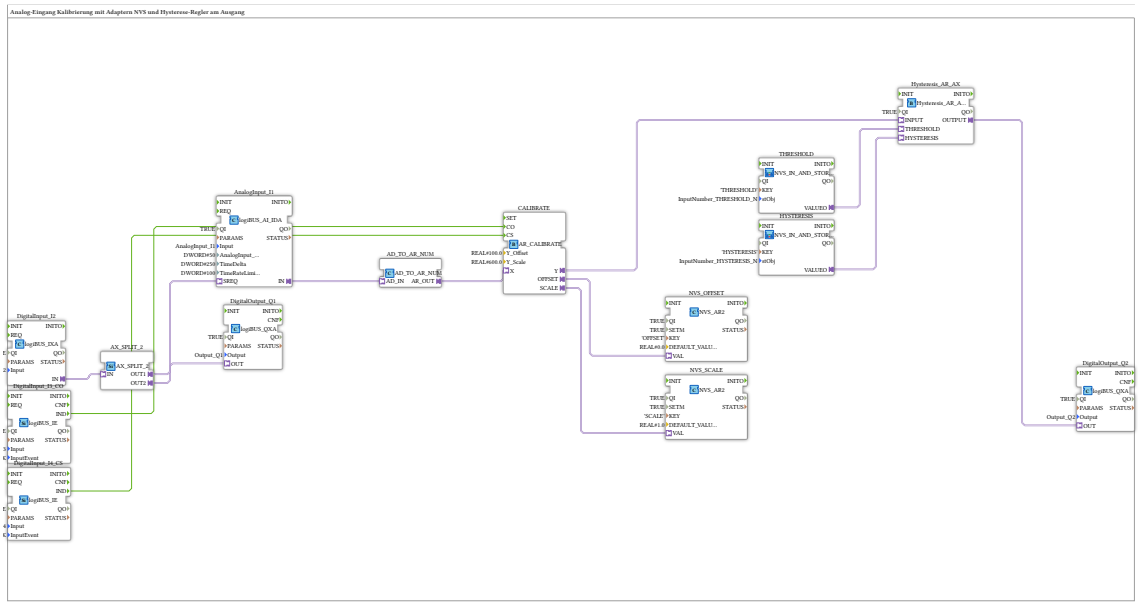
Zusammenfassung

Die Übung 028a_AX zeigt eine Kalibrierung eines Analog-Eingangs mit AR_CALIBRATE, bei der Messwert und Kalibrierwerte ausschließlich über Adapter geführt werden. Wichtig ist die numerische Konvertierung mit AD_TO_AR_NUM (AD → UDINT → REAL); ein einfaches AD_TO_AR würde den Wert nur umdeuten („reinterpret_cast“). OFFSET und SCALE werden über INI_AR2 in der INI-Sektion Uebung_028a_AX abgelegt.

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

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

2.1.2.266 Uebung_028b2_AR: Analog-Eingang Kalibrierung mit Adaptern NVS und Hysterese-Regler am Ausgang



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung realisiert eine analoge Eingangskalibrierung mit Offset- und Skalierungsanpassung. Die Kalibrierungsparameter werden persistent im NVS (Non-Volatile Storage) gespeichert. Zusätzlich wird ein Hysterese-Regler verwendet, der auf den kalibrierten Analogwert reagiert. Die Schwellwerte und Hysteresebänder für den Regler werden ebenfalls im NVS gehalten und lassen sich über Eingabefelder im VT ändern. Der Ablauf wird durch digitale Eingänge gesteuert und über digitale Ausgänge ausgegeben.

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_I2**: Digitaler Eingang
  - **Typ**: `logiBUS::io::DI::logiBUS_IXA`
  - **Parameter**: `QI = TRUE`, `Input = Input_I2`
  - **Erklärung**: Liefert den Zustand von `I2` als AX-Adapterausgang `IN`.
- **AX_SPLIT_2**: Signalverteiler
  - **Typ**: `adapter::events::unidirectional::AX_SPLIT_2`
  - **Parameter**: keine
  - **Erklärung**: Verteilt das Signal von `DigitalInput_I2` auf zwei Wege: `OUT1` an `AnalogInput_I1.SREQ`, `OUT2` zum Digitalausgang.
- **DigitalOutput_Q1**: Digitaler Ausgang zur direkten Weitergabe des Eingangsstatus
  - **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
  - **Parameter**: `QI = TRUE`, `Output = Output_Q1`
  - **Erklärung**: Schaltet `Q1` gemäß `I2`.
- **AnalogInput_I1**: Analoger Eingang mit Hysterese und Ratenbegrenzung
  - **Typ**: `logiBUS::io::AI::logiBUS_AI_IDA`
  - **Parameter**: `QI = TRUE`, `Input = AnalogInput_I1`, `AnalogInput_hysteresis = 50`, `TimeDelta = 250`, `TimeRateLimit = 100`
  - **Erklärung**: Liefert den Analogwert als Adapter `IN` (DWORD-Adapter `AD`); `SREQ` wird von `I2` über `AX_SPLIT_2` versorgt.
- **AD_TO_AR_NUM**: Adapterkonverter AD ⟶ REAL (numerisch)
  - **Typ**: `adapter::conversion::unidirectional::AD_TO_AR_NUM`
  - **Parameter**: keine
  - **Erklärung**: Wandelt den `AD`-Adapter (DWORD) numerisch (AD ⟶ UDINT ⟶ REAL) in einen `AR`-Adapter (REAL) um. Ein `AD_TO_AR` würde wie ein `reinterpret_cast` (Bitmuster) wirken (Kommentar im Netzwerk).
- **DigitalInput_I3_CO**: Eingang zum Auslösen der Offset-Kalibrierung

  - **Typ**: `logiBUS::io::DI::logiBUS_IE`
  - **Parameter**: `QI = TRUE`, `Input = Input_I3`, `InputEvent = BUTTON_SINGLE_CLICK`
  - **Erklärung**: Ein Einzelklick auf `I3` löst `CALIBRATE.CO` aus.
- **DigitalInput_I4_CS**: Eingang zum Auslösen der Skalierungskalibrierung
  - **Typ**: `logiBUS::io::DI::logiBUS_IE`
  - **Parameter**: `QI = TRUE`, `Input = Input_I4`, `InputEvent = BUTTON_SINGLE_CLICK`
  - **Erklärung**: Ein Einzelklick auf `I4` löst `CALIBRATE.CS` aus.
- **CALIBRATE**: Zwei-Punkt-Kalibrierung
  - **Typ**: `adapter::Engineering::measurements::AR_CALIBRATE`
  - **Parameter**: `Y_Offset = 100.0`, `Y_Scale = 600.0`
  - **Erklärung**: Berechnet `Y = (X + OFFSET) * SCALE`. `CO` setzt `OFFSET := Y_Offset - X`, `CS` setzt `SCALE := Y_Scale / (X + OFFSET)`. `CO` muss vor `CS` ausgeführt werden. `Y_Offset` und `Y_Scale` sind Zielwerte auf der Ausgangsseite.
- **NVS_OFFSET**: persistenter Speicher für den Offset
  - **Typ**: `logiBUS::storage::esp32_nvs::NVS_AR2`
  - **Parameter**: `QI = TRUE`, `SETM = TRUE`, `KEY = 'OFFSET'`, `DEFAULT_VALUE = 0.0`
  - **Erklärung**: Lädt und speichert den Offset unter dem Schlüssel `OFFSET` über den bidirektionalen Adapter `VAL` (verbunden mit `CALIBRATE.OFFSET`).
- **NVS_SCALE**: persistenter Speicher für die Skalierung
  - **Typ**: `logiBUS::storage::esp32_nvs::NVS_AR2`
  - **Parameter**: `QI = TRUE`, `SETM = TRUE`, `KEY = 'SCALE'`, `DEFAULT_VALUE = 1.0`
  - **Erklärung**: Lädt und speichert den Skalierungsfaktor unter dem Schlüssel `SCALE` (verbunden mit `CALIBRATE.SCALE`).
- **Hysteresis_AR_AX**: Hysterese-Komparator
  - **Typ**: `logiBUS::signalprocessing::hysteresis::Hysteresis_AR_AX`
  - **Parameter**: `QI = TRUE`
  - **Erklärung**: Wandelt `INPUT` (AR) in ein digitales Signal `OUTPUT` (AX): Einschalten bei `THRESHOLD + HYSTERESIS / 2`, Ausschalten bei `THRESHOLD - HYSTERESIS / 2`.
- **DigitalOutput_Q2**: Digitaler Ausgang für das Hysteresesignal
  - **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
  - **Parameter**: `QI = TRUE`, `Output = Output_Q2`
  - **Erklärung**: Schaltet `Q2` gemäß `Hysteresis_AR_AX.OUTPUT`.

Sub-Bausteine: `THRESHOLD` und `HYSTERESIS`

Beide sind Instanzen von `MyLib::sys::NVS_IN_AND_STORE_AR` (in `MyLib_AX`). Intern besteht der Baustein aus `NumericValue_PHYSA` (VT-Eingabefeld, liefert `rPhys`), `NVS_AR` (`QI = TRUE`, `SETM = TRUE`, `DEFAULT_VALUE = REAL#0.0`; Laden und Speichern unter `KEY`), `AR_SPLIT_2` und `Q_NumericValue_PHYSA` (schreibt den Wert zurück in das VT-Feld). Der Ausgang `VALUEO` ist ein `AR`-Adapter.

- **THRESHOLD**: `KEY = 'THRESHOLD'`, `stObj = InputNumber_THRESHOLD_N`. Liefert den Schwellwert (Mitte des Hysteresebandes) an `Hysteresis_AR_AX.THRESHOLD`.
- **HYSTERESIS**: `KEY = 'HYSTERESIS'`, `stObj = InputNumber_HYSTERESIS_N`. Liefert die volle Breite des Hysteresebandes an `Hysteresis_AR_AX.HYSTERESIS`.

Programmablauf und Verbindungen

1. `DigitalInput_I2.IN` → `AX_SPLIT_2.IN`: `AX_SPLIT_2.OUT1` → `AnalogInput_I1.SREQ` und `AX_SPLIT_2.OUT2` → `DigitalOutput_Q1.OUT` (direkte Ausgabe).
2. `AnalogInput_I1.IN` → `AD_TO_AR_NUM.AD_IN`, `AD_TO_AR_NUM.AR_OUT` → `CALIBRATE.X`: der Analogwert wird numerisch als REAL in die Kalibrierung gegeben.
3. `DigitalInput_I3_CO.IND` → `CALIBRATE.CO` und `DigitalInput_I4_CS.IND` → `CALIBRATE.CS`: Offset- bzw. Skalierungskalibrierung per Einzelklick.
4. `CALIBRATE.OFFSET` → `NVS_OFFSET.VAL` und `CALIBRATE.SCALE` → `NVS_SCALE.VAL`: die Kalibrierwerte werden im NVS abgelegt und geladen.
5. `CALIBRATE.Y` → `Hysteresis_AR_AX.INPUT`, `THRESHOLD.VALUEO` → `Hysteresis_AR_AX.THRESHOLD`, `HYSTERESIS.VALUEO` → `Hysteresis_AR_AX.HYSTERESIS`.
6. `Hysteresis_AR_AX.OUTPUT` → `DigitalOutput_Q2.OUT`: das Ergebnis des Komparators schaltet `Q2`.

Lernziele:

- Verständnis der Analogwertaufbereitung mit Adaptern und Konvertierungen in 4diac.
- Implementierung einer persistenten Kalibrierung (Offset und Skalierung) im NVS.

- Anwendung eines Hysterese-Reglers mit extern vorgebbaren Parametern.
- Steuerung des Ablaufs durch digitale Eingänge.

Schwierigkeitsgrad: Fortgeschritten **Vorkenntnisse**: Grundlagen der 4diac-IDE, Umgang mit analogen Ein-/Ausgängen, einfache Adapter und NVS-Speicher.

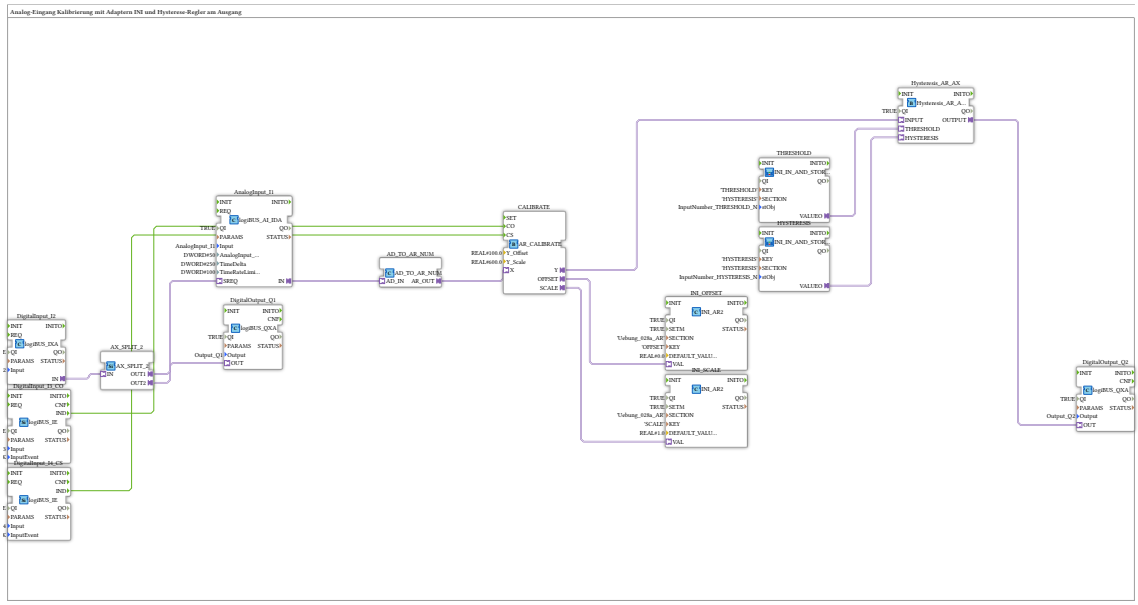
Zusammenfassung

Die Übung zeigt eine vollständige analoge Messkette: vom Einlesen des rohen Analogwerts über die Kalibrierung mit persistenter Speicherung bis hin zur Ausgabe über einen Hysterese-Komparator. Die Verwendung von Adaptern zur Typkonvertierung und von Sub-Bausteinen zur Wiederverwendung von NVS-Zugriffen macht den Aufbau modular und erweiterbar. Die Kalibrierung kann jederzeit über digitale Taster angepasst werden, ohne dass der Programmcode geändert werden muss.

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

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

2.1.2.267 Uebung_028b_AR: Analog-Eingang Kalibrierung mit Adaptern INI und Hysterese-Regler am Ausgang



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert die Kalibrierung eines analogen Eingangssignals mit Hilfe von Adaptern und einer INI-basierten Speicherung der Kalibrierparameter. Das kalibrierte Signal wird anschließend durch einen Hysterese-Regler geführt, dessen Schwellwert und Hysterese über das VT eingegeben und ebenfalls in der INI-Datei gespeichert werden. Die Übung zeigt die Verknüpfung von analogen und digitalen Ein-/Ausgängen, Adapter-Konvertierungen sowie die persistente Speicherung von Parametern.

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_I2**: logiBUS Digitaleingang
  - **Typ**: `logiBUS::io::DI::logiBUS_IXA`
  - **Parameter**: `QI = TRUE`, `Input = Input_I2`
  - **Erklärung**: Liefert den Zustand von `I2` als AX-Adaptersignal.
- **AX_SPLIT_2**: Signalverteiler
  - **Typ**: `adapter::events::unidirectional::AX_SPLIT_2`
  - **Parameter**: keine
  - **Erklärung**: Verteilt das AX-Signal von `I2` auf zwei Ausgänge: `OUT1` zum Analogeingang (Messanforderung), `OUT2` zum Digitalausgang `Q1`.
- **DigitalOutput_Q1**: logiBUS Digitalausgang

  - **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
  - **Parameter**: `QI = TRUE`, `Output = Output_Q1`
  - **Erklärung**: Gibt das Signal von `I2` am physischen Ausgang `Q1` aus.
- **AnalogInput_I1**: logiBUS Analogeingang
  - **Typ**: `logiBUS::io::AI::logiBUS_AI_IDA`
  - **Parameter**: `QI = TRUE`, `Input = AnalogInput_I1`, `AnalogInput_hysteresis = 50`, `TimeDelta = 250`, `TimeRateLimit = 100`
  - **Erklärung**: Stellt den analogen Messwert am AD-Adapter `IN` bereit. Über den Socket `SREQ` kann eine Messung extern angefordert werden.
- **AD_TO_AR_NUM**: Konvertierungsadapter
  - **Typ**: `adapter::conversion::unidirectional::AD_TO_AR_NUM`
  - **Parameter**: keine
  - **Erklärung**: Wandelt den Rohwert vom AD-Adapter numerisch (AD → UDINT → REAL) in einen REAL-Wert am AR-Adapter. **Hinweis** (Kommentar im Netzwerk): Ein `AD_TO_AR` wäre wie ein „reinterpret_cast“ (Bitmuster); `AD_TO_AR_NUM` wandelt dagegen den Zahlenwert.
- **DigitalInput_I3_CO**: Klick-Eingang für die Offset-Kalibrierung
  - **Typ**: `logiBUS::io::DI::logiBUS_IE`
  - **Parameter**: `QI = TRUE`, `Input = Input_I3`, `InputEvent = BUTTON_SINGLE_CLICK`
  - **Erklärung**: Meldet einen Einzelklick auf `I3` als Ereignis `IND`.
- **DigitalInput_I4_CS**: Klick-Eingang für die Skalen-Kalibrierung
  - **Typ**: `logiBUS::io::DI::logiBUS_IE`
  - **Parameter**: `QI = TRUE`, `Input = Input_I4`, `InputEvent = BUTTON_SINGLE_CLICK`
  - **Erklärung**: Meldet einen Einzelklick auf `I4` als Ereignis `IND`.
- **CALIBRATE**: Kalibrierung
  - **Typ**: `adapter::Engineering::measurements::AR_CALIBRATE`
  - **Parameter**: `Y_Offset = 100.0`, `Y_Scale = 600.0`
  - **Erklärung**: Berechnet `Y = (X + OFFSET) * SCALE`. Das Ereignis `CO` ermittelt den Offset als `Y_Offset - X`, das Ereignis `CS` den Skalenfaktor als `Y_Scale / (X + OFFSET)`. Ausgänge sind der kalibrierte Wert `Y` sowie die Adapter `OFFSET` und `SCALE`.
- **INI_OFFSET**: INI-Speicher für den Offset
  - **Typ**: `eclipse4diac::storage::INI_AR2`
  - **Parameter**: `QI = TRUE`, `SETM = TRUE`, `SECTION = 'Uebung_028a_AR'`, `KEY = 'OFFSET'`, `DEFAULT_VALUE = 0.0`
  - **Erklärung**: Liest den Wert aus der INI-Datei und schreibt über den bidirektionalen Adapter `VAL` den von `CALIBRATE` berechneten Offset zurück.
- **INI_SCALE**: INI-Speicher für die Skalierung
  - **Typ**: `eclipse4diac::storage::INI_AR2`
  - **Parameter**: `QI = TRUE`, `SETM = TRUE`, `SECTION = 'Uebung_028a_AR'`, `KEY = 'SCALE'`, `DEFAULT_VALUE = 1.0`
  - **Erklärung**: Wie `INI_OFFSET`, jedoch für den Skalenfaktor.
- **THRESHOLD**: VT-Eingabe und Speicherung des Schwellwerts
  - **Typ**: `MyLib::sys::INI_IN_AND_STORE_AR` (SubApp-Instanz)
  - **Parameter**: `KEY = 'THRESHOLD'`, `SECTION = 'HYSTERESIS'`, `stObj = InputNumber_THRESHOLD_N`
  - **Erklärung**: Ein über das VT (Eingabefeld `InputNumber_THRESHOLD_N`) eingegebener, physikalisch skalierter REAL-Wert wird in der INI-Datei abgelegt, wieder auf dem VT angezeigt und am Adapter `VALUEO` ausgegeben.
- **HYSTERESIS**: VT-Eingabe und Speicherung der Hysteresebreite
  - **Typ**: `MyLib::sys::INI_IN_AND_STORE_AR` (SubApp-Instanz)
  - **Parameter**: `KEY = 'HYSTERESIS'`, `SECTION = 'HYSTERESIS'`, `stObj = InputNumber_HYSTERESIS_N`
  - **Erklärung**: Wie `THRESHOLD`, jedoch für die Hysteresebreite (Eingabefeld `InputNumber_HYSTERESIS_N`).
- **Hysteresis_AR_AX**: Hysterese-Regler
  - **Typ**: `logiBUS::signalprocessing::hysteresis::Hysteresis_AR_AX`
  - **Parameter**: `QI = TRUE`
  - **Erklärung**: Schaltet `OUTPUT` ein, wenn `INPUT` den Wert `THRESHOLD + HYSTERESIS/2` erreicht oder überschreitet, und aus, wenn `INPUT` den Wert `THRESHOLD - HYSTERESIS/2` unterschreitet.
- **DigitalOutput_Q2**: logiBUS Digitalausgang
  - **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
  - **Parameter**: `QI = TRUE`, `Output = Output_Q2`

- **Erklärung**: Gibt das Ergebnis des Hysterese-Reglers am physischen Ausgang `Q2` aus.

Programmablauf und Verbindungen

1. `DigitalInput_I2.IN` ⟶ `AX_SPLIT_2.IN` (AdapterConnection): Der Zustand von `I2` wird verteilt.
2. `AX_SPLIT_2.OUT2` ⟶ `DigitalOutput_Q1.OUT`: `Q1` folgt `I2`.
3. `AX_SPLIT_2.OUT1` ⟶ `AnalogInput_I1.SREQ`: `I2` fordert eine Messung am Analogeingang an.
4. `AnalogInput_I1.IN` ⟶ `AD_TO_AR_NUM.AD_IN`: Der Messwert wird numerisch in einen REAL-Wert umgewandelt.
5. `AD_TO_AR_NUM.AR_OUT` ⟶ `CALIBRATE.X`: Der umgewandelte Wert ist der Kalibriereingang.
6. `DigitalInput_I3_CO.IND` ⟶ `CALIBRATE.CO` und `DigitalInput_I4_CS.IND` ⟶ `CALIBRATE.CS` (Event-Connections): Ein Klick auf `I3` löst die Offset-Kalibrierung aus, ein Klick auf `I4` die Skalen-Kalibrierung.
7. `CALIBRATE.OFFSET` ⟶ `INI_OFFSET.VAL` und `CALIBRATE.SCALE` ⟶ `INI_SCALE.VAL`: Die berechneten Kalibrierwerte werden in der INI-Datei (Sektion `Uebung_028a_AR`) gespeichert.
8. `CALIBRATE.Y` ⟶ `Hysteresis_AR_AX.INPUT`: Der kalibrierte Messwert wird ausgewertet.
9. `THRESHOLD.VALUEO` ⟶ `Hysteresis_AR_AX.THRESHOLD` und `HYSTERESIS.VALUEO` ⟶ `Hysteresis_AR_AX.HYSTERESIS`: Schwellwert und Hysteresebreite aus den VT-Eingaben.
10. `Hysteresis_AR_AX.OUTPUT` ⟶ `DigitalOutput_Q2.OUT`: Das Hysterese-Ergebnis schaltet `Q2`.

Lernziele:

- Verständnis der Adapter-Konvertierung zwischen analogen Signaltypen (`AD` ⟶ `AR`, numerisch mit `AD_TO_AR_NUM`)
- Umgang mit **INI**-Bausteinen zum persistenten Speichern und Laden von Kalibrierparametern
- Einsatz eines Kalibrierungsadapters (Offset/Skalierung)
- Realisierung einer Hysterese-Funktion zur Schwellwertüberwachung

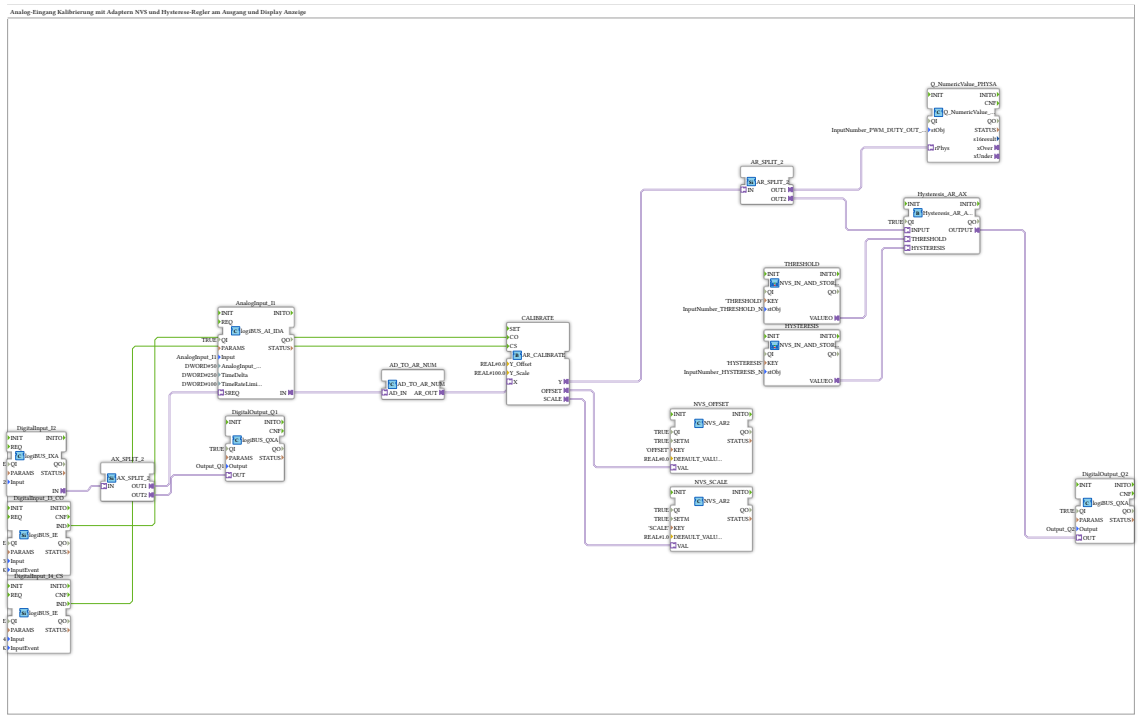
Zusammenfassung

Die Übung **Uebung_028b_AR** realisiert eine vollständige Kette zur Verarbeitung eines analogen Eingangssignals: Messung, Kalibrierung, Speicherung der Kalibrierdaten und anschließende Hysterese-Auswertung. Durch die Kombination von digitalen Ereignissen, Adapter-Konvertierungen und INI-basierter Parameterverwaltung wird ein praxisnahes Beispiel für industrielle Analogwertverarbeitung in 4diac dargestellt.

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

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

2.1.2.268 Uebung_028c2_AR: Analog-Eingang Kalibrierung mit Adaptern NVS und Hysterese-Regler am Ausgang und Display Anzeige



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert die Kalibrierung eines analogen Eingangssignals unter Verwendung von NVS-Speicher (Non-Volatile Storage) für Offset und Skalierung. Das kalibrierte Signal wird auf zwei Wege aufgeteilt: Einerseits zur Anzeige eines physikalischen Werts (z. B. für ein Display), andererseits zu einem Hysterese-Regler, der einen digitalen Ausgang ansteuert. Die Kalibrierung kann über digitale Eingänge (Offset- und Skalierungsbefehle) gestartet werden. Die Schwellwerte für die Hysterese werden aus dem NVS über zwei Sub-Applikationen geladen.

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: `THRESHOLD` und `HYSTERESIS`

- **Typ**: `MyLib::sys::NVS_IN_AND_STORE_AR`
- **Beschreibung**: Beide Sub-Applikationen speichern einen über das VT eingegebenen, physikalisch skalierten REAL-Wert (AR-Adapter) persistent im NVS des ESP32. Der Wert wird über den Parameter `KEY` adressiert (`'THRESHOLD'` bzw. `'HYSTERESIS'`), das zugehörige VT-Eingabeobjekt über den Parameter `stObj` (`InputNumber_THRESHOLD_N` bzw. `InputNumber_HYSTERESIS_N`). Der Ausgang `VALUEO` liefert den Wert.

Übersicht aller verwendeten Funktionsbausteine

Name	Typ	Parameter (Auswahl)
`DigitalInput_I2`	`logiBUS::io::DI::logiBUS_IXA`	`QI=TRUE`, `Input = Input_I2`
`DigitalOutput_Q1`	`logiBUS::io::DQ::logiBUS_QXA`	`QI=TRUE`, `Output = Output_Q1`
`AnalogInput_I1`	`logiBUS::io::AI::logiBUS_AI_IDA`	`QI=TRUE`, `Input = AnalogInput_I1`, `AnalogInput_hysteresis=50`, `TimeDelta=250`, `TimeRateLimit=100`
`CALIBRATE`	`Adapter::Engineering::measurements::AR_CALIBRATE`	`Y_Offset=0.0`, `Y_Scale=100.0`

NVS_OFFSET	logiBUS::storage::esp32_nvs::NVS_AR2	QI=TRUE, SETM=TRUE, KEY='OFFSET', DEFAULT_VALUE=0.0
NVS_SCALE	logiBUS::storage::esp32_nvs::NVS_AR2	QI=TRUE, SETM=TRUE, KEY='SCALE', DEFAULT_VALUE=1.0
DigitalInput_I3_CO	logiBUS::io::DI::logiBUS_IE	Input = Input_I3, InputEvent = BUTTON_SINGLE_CLICK (Kalibrier-Offset-Befehl)
DigitalInput_I4_CS	logiBUS::io::DI::logiBUS_IE	Input = Input_I4, InputEvent = BUTTON_SINGLE_CLICK (Kalibrier-Skalierungsbefehl)
AX_SPLIT_2	adapter::events::unidirectional::AX_SPLIT_2	–
THRESHOLD	SubApp MyLib::sys::NVS_IN_AND_STORE_AR	KEY='THRESHOLD', stObj=InputNumber_THRESHOLD_N
HYSTERESIS	SubApp MyLib::sys::NVS_IN_AND_STORE_AR	KEY='HYSTERESIS', stObj=InputNumber_HYSTERESIS_N
Hysteresis_AR_AX	logiBUS::signalprocessing::hysteresis::Hysteresis_AR_AX	QI=TRUE
AR_SPLIT_2	adapter::events::unidirectional::AR_SPLIT_2	–
Q_NumericValue_PHYSA	isobus::UT::Q::Q_NumericValue_PHYSA	stObj=InputNumber_PWM_DUTY_OUT_N (Anzeige)
AD_TO_AR_NUM	adapter::conversion::unidirectional::AD_TO_AR_NUM	–
DigitalOutput_Q2	logiBUS::io::DQ::logiBUS_QXA	QI=TRUE, Output = Output_Q2

Kurzbeschreibung der wichtigsten Komponenten

- **AnalogInput_I1**: Liest den analogen Eingang `AnalogInput_I1` (z. B. Spannung) ein und gibt ihn als Adapter-Interface (`AD`) aus. Die Abtastung wird über `SREQ` angefordert.
- **AD_TO_AR_NUM**: Wandelt das analoge Adapter-Interface (AD) numerisch (AD ⟶ UDINT ⟶ REAL) in einen analogen Real-Wert (`AR`) um.
- **CALIBRATE**: Wendet Offset und Skalierung auf den eingehenden analogen Wert an: `Y = (X + Offset) * Scale`. Die Werte für Offset und Skalierung können über die Ereigniseingänge `CO` und `CS` aktualisiert und anschließend in den NVS-Bausteinen gespeichert werden. Die Referenzwerte `Y_Offset = 0.0` und `Y_Scale = 100.0` entsprechen laut Netzwerk-Kommentar dem Beispiel 0,0 ⟹ 0,5 V und 100,0 ⟹ 4,5 V (typischer Automotive-Sensor).
- **NVS_OFFSET, NVS_SCALE**: Speichern die Kalibrierwerte dauerhaft im Flash des ESP32. Der Ausgang `VAL` liefert den aktuell gespeicherten Wert.
- **Hysteresis_AR_AX**: Vergleicht den kalibrierten Wert mit einem Schwellwert und einer Hysterese. Der Ausgang `OUTPUT` (⟶ `DigitalOutput_Q2.OUT`) schaltet, wenn der Wert den Schwellwert überschreitet (bzw. unterschreitet inkl. Hysterese).
- **Q_NumericValue_PHYSA**: Bereitet den kalibrierten Wert zur Anzeige auf einem Display oder einer anderen Ausgabeeinheit auf.
- **AX_SPLIT_2, AR_SPLIT_2**: Verteilen ein Signal (Event bzw. Daten) auf zwei Ausgänge.

Programmablauf und Verbindungen

1. **Analoger Eingang**: Der Analogbaustein `AnalogInput_I1` (logiBUS AI) liefert einen analogen Messwert als `AD`-Adapter.
2. **Konvertierung**: Über `AD_TO_AR_NUM` wird der Wert numerisch in einen `AR`-Real-Wert umgewandelt. Ein Kommentar weist darauf hin, dass ein `AD_TO_AR` wie ein `reinterpret_cast` (Bitmuster) wirken würde – `AD_TO_AR_NUM` wandelt dagegen den Zahlenwert (AD ⟶ UDINT ⟶ REAL).
3. **Kalibrierung**: Der `AR`-Wert wird an `CALIBRATE.X` übergeben. Die digitalen Eingänge `I3` (CO) und `I4` (CS) lösen die Berechnung von Offset (`CO`-Ereignis) und Skalierung (`CS`-Ereignis) aus. Die berechneten Werte werden über die NVS-Bausteine gespeichert.
   - `DigitalInput_I3_CO` ⟶ `CALIBRATE.CO` (Offset ermitteln)
   - `DigitalInput_I4_CS` ⟶ `CALIBRATE.CS` (Skalierung ermitteln)

4. **Aufteilung des kalibrierten Werts**: Der Ausgang `CALIBRATE.Y` wird über `AR_SPLIT_2` auf zwei Pfade verteilt:
   - Pfad 1: Anzeige ⟶ `Q_NumericValue_PHYSA.rPhys` (`InputNumber_PWM_DUTY_OUT_N`)
   - Pfad 2: Hysterese ⟶ `Hysteresis_AR_AX.INPUT`
5. **Hysterese**: Die Sub-Applikationen `THRESHOLD` und `HYSTERESIS` liefern die Schwellwerte (`THRESHOLD.VALUEO` ⟶ `Hysteresis_AR_AX.THRESHOLD` und `HYSTERESIS.VALUEO` ⟶ `Hysteresis_AR_AX.HYSTERESIS`). Der Hysterese-Baustein vergleicht den Eingang mit diesen Werten und schaltet seinen Ausgang `OUTPUT`.
6. **Digitale Ausgänge**:
   - `DigitalInput_I2` wird über `AX_SPLIT_2` verteilt: `OUT1` ⟶ `AnalogInput_I1.SREQ` (löst die analoge Abtastung aus), `OUT2` ⟶ `DigitalOutput_Q1.OUT` (der Ausgang `Q1` folgt dem Eingang `I2`).
   - `Hysteresis_AR_AX.OUTPUT` schaltet den digitalen Ausgang `DigitalOutput_Q2` (z. B. für eine Schaltfunktion).

Wichtiger Hinweis: Ein `AD_TO_AR` wäre wie ein „reinterpret_cast“ (Bitmuster). `AD_TO_AR_NUM` wandelt dagegen den Zahlenwert (AD ⟶ UDINT ⟶ REAL) und gewährleistet so eine korrekte Werteübertragung (siehe Kommentar im Netzwerk).

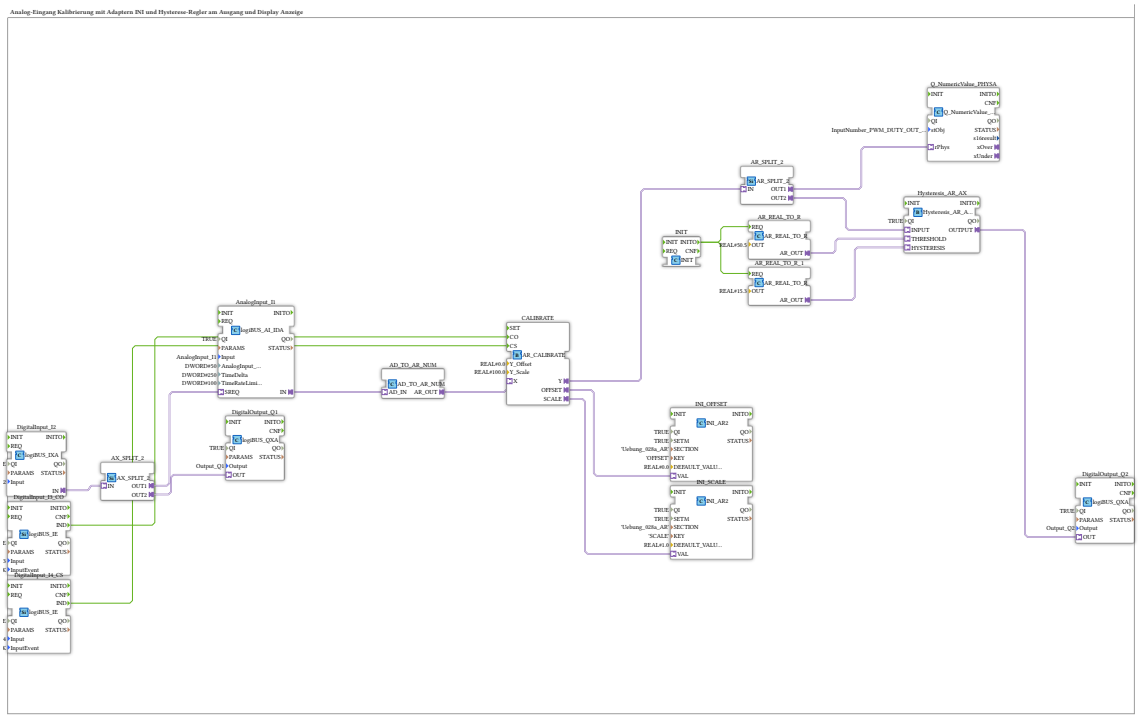
Zusammenfassung

Die Übung zeigt, wie ein analoges Eingangssignal mit Offset- und Skalierungskorrektur kalibriert wird. Die Kalibrierparameter werden dauerhaft im NVS gespeichert und können über digitale Taster aktualisiert werden. Der kalibrierte Wert wird sowohl für eine Anzeige als auch für eine Hysterese-Schaltfunktion genutzt. Die Verschaltung verdeutlicht den Umgang mit Adapter-Konvertierungen, NVS-Speicherzugriffen und der Aufteilung von Datenflüssen in der 4diac-IDE.

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

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de
- 🌐 ESP32 & ESP32-S3 DevKit auf ms-muc-docs.de
- 🌐 Das PWM-Signal & Infografik auf ms-muc-docs.de

2.1.2.269 Uebung_028c3_AR: Analog-Eingang Kalibrierung mit Adaptern INI und Hysterese-Regler am Ausgang und Display Anzeige



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert die Kalibrierung eines analogen Eingangs (AnalogInput_I1) mithilfe eines Kalibrierungsadapters (AR_CALIBRATE), der einen Offset und einen Skalierungsfaktor aus einer INI-Datei lädt. Das kalibrierte Signal wird an einen Hysterese-Regler (Hysteresis_AR_AX) übergeben, der einen digitalen Ausgang (Output_Q2) schaltet. Gleichzeitig wird der kalibrierte Wert auf einem Display (Q_NumericValue_PHYSA) angezeigt. Die Übung nutzt zwei Eingänge (I3, I4) zum Triggern der Kalibrierung und einen weiteren Eingang (I2) als Freigabesignal.

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: Keine (alle FBs sind systemdefiniert)

- **AnalogInput_I1** (Typ: `logiBUS::io::AI::logiBUS_AI_IDA`)
  - **Verwendete interne Parameter**: `AnalogInput_hysteresis = 50`, `TimeDelta = 250`, `TimeRateLimit = 100`
  - **Funktionsweise**: Liest den physikalischen Analogwert (z. B. Spannung) vom Eingang I1 aus. Die Parameter legen eine Hysterese, Abtastzeit und Ratenbegrenzung fest.
- **DigitalInput_I2** (Typ: `logiBUS::io::DI::logiBUS_IXA`)
  - **Funktionsweise**: Liest den digitalen Eingang I2. Wird als Freigabesignal für den Ausgang Q1 verwendet.
- **DigitalInput_I3_CO** (Typ: `logiBUS::io::DI::logiBUS_IE`)
  - **Parameter**: `QI = TRUE`, `Input = Input_I3`, `InputEvent = BUTTON_SINGLE_CLICK`
  - **Funktionsweise**: Ein Einzelklick an I3 erzeugt ein Ereignis an `DigitalInput_I3_CO.IND`. Die Verbindung `DigitalInput_I3_CO.IND` → `CALIBRATE.CO` löst den Kalibrier-Commit aus.
- **DigitalInput_I4_CS** (Typ: `logiBUS::io::DI::logiBUS_IE`)
  - **Parameter**: `QI = TRUE`, `Input = Input_I4`, `InputEvent = BUTTON_SINGLE_CLICK`
  - **Funktionsweise**: Ein Einzelklick an I4 erzeugt ein Ereignis an `DigitalInput_I4_CS.IND`. Die Verbindung `DigitalInput_I4_CS.IND` → `CALIBRATE.CS` startet das Lesen der Kalibrierdaten.
- **DigitalOutput_Q1** (Typ: `logiBUS::io::DQ::logiBUS_QXA`)
  - **Funktionsweise**: Setzt den digitalen Ausgang Q1 basierend auf dem Eingang I2 (Freigabe).
- **DigitalOutput_Q2** (Typ: `logiBUS::io::DQ::logiBUS_QXA`)
  - **Funktionsweise**: Setzt den digitalen Ausgang Q2 basierend auf dem Hysterese-Regler-Ausgang.
- **CALIBRATE** (Typ: `adapter::Engineering::measurements::AR_CALIBRATE`)
  - **Verwendete interne Parameter**: `Y_Offset = 0.0`, `Y_Scale = 100.0`
  - **Funktionsweise**: Führt eine Kalibrierung des analogen Eingangswertes durch. Berechnet einen Offset- und Skalierungsfaktor, der in die INI-Datei geschrieben wird. Eingänge: X (Rohwert), CO (Commit-Trigger), CS (Read-Trigger). Ausgänge: Y (kalibrierter Wert), OFFSET, SCALE.
- **INI_OFFSET** (Typ: `eclipse4diac::storage::INI_AR2`)
  - **Verwendete interne Parameter**: `SECTION = 'Uebung_028a_AR'`, `KEY = 'OFFSET'`, `DEFAULT_VALUE = 0.0`
  - **Funktionsweise**: Liest den Offset-Wert aus einer INI-Datei und stellt ihn als analogen Wert bereit.
- **INI_SCALE** (Typ: `eclipse4diac::storage::INI_AR2`)
  - **Verwendete interne Parameter**: `SECTION = 'Uebung_028a_AR'`, `KEY = 'SCALE'`, `DEFAULT_VALUE = 1.0`
  - **Funktionsweise**: Liest den Skalierungsfaktor aus einer INI-Datei und stellt ihn als analogen Wert bereit.
- **Hysteresis_AR_AX** (Typ: `logiBUS::signalprocessing::hysteresis::Hysteresis_AR_AX`)
  - **Verwendete interne Parameter**: `QI = TRUE`
  - **Funktionsweise**: Vergleicht den kalibrierten Eingangswert mit einem Schwellwert und einer Hysterese. Der Ausgang wird gesetzt, wenn der Wert den Schwellwert überschreitet und zurückgesetzt, wenn er unter den unteren Schwellwert (Schwellwert – Hysterese/2) fällt; der obere Schwellwert ist Schwellwert + Hysterese/2.
- **Q_NumericValue_PHYSA** (Typ: `isobus::UT::Q::Q_NumericValue_PHYSA`)
  - **Verwendete interne Parameter**: `stObj = InputNumber_PWM_DUTY_OUT_N`
  - **Funktionsweise**: Zeigt einen analogen Wert auf einem Display an (z. B. als numerischen Wert).
- **AX_SPLIT_2** (Typ: `adapter::events::unidirectional::AX_SPLIT_2`)
  - **Funktionsweise**: Verteilt ein digitales Signal (AX) auf zwei Ausgänge (OUT1, OUT2). Hier: Eingang von DigitalInput_I2 wird geteilt als Trigger für AnalogInput_I1 (OUT1) und an DigitalOutput_Q1 (OUT2).
- **AR_SPLIT_2** (Typ: `adapter::events::unidirectional::AR_SPLIT_2`)

  - **Funktionsweise**: Verteilt einen analogen Wert (AR) auf zwei Ausgänge (OUT1, OUT2). Hier: Kalibrierter Wert geht an Display und Hysterese-Regler.
- **AD_TO_AR_NUM** (Typ: `adapter::conversion::unidirectional::AD_TO_AR_NUM`)
  - **Funktionsweise**: Konvertiert einen AD-Wert (Analogwert-Adapter) numerisch (AD → UDINT → REAL) in einen AR-Wert (analoger realer Wert). Hinweis: Ein direktes AD_TO_AR würde wie ein „reinterpret_cast“ (Bitmuster) wirken.
- **AR_REAL_TO_R** (x2) (Typ: `adapter::conversion::unidirectional::AR_REAL_TO_R`)
  - **Verwendete interne Parameter**: Erstes: `OUT = 50.5`, Zweites: `OUT = 15.3`
  - **Funktionsweise**: Wandelt einen konstanten realen Wert in ein AR-Signal um. Dient als Schwellwert (THRESHOLD = 50.5) und Hysteresebreite (HYSTERESIS = 15.3) für den Hysterese-Regler.
- **INIT** (Typ: `iec61131::booleanOperators::INIT`)
  - **Funktionsweise**: Erzeugt ein einmaliges Initialisierungsereignis (INITO) beim Systemstart. Dieses Ereignis triggert die beiden AR_REAL_TO_R-Bausteine, um die konstanten Werte zu setzen.

Programmablauf und Verbindungen

1. **Initialisierung**: Beim Start triggert der INIT-Baustein die beiden AR_REAL_TO_R-Bausteine, die die konstanten Schwellwerte (50.5 und 15.3) auf den analogen Bus legen.
2. **Analogwerterfassung**: Der AnalogInput_I1 liest den Rohwert vom Eingang I1. Dieser Rohwert wird über den Konverter AD_TO_AR_NUM numerisch in einen REAL-Wert gewandelt und an den Kalibrierungsbaustein CALIBRATE übergeben.
3. **Kalibrierung**: Die digitalen Eingänge I3 (CO) und I4 (CS) steuern die Kalibrierung:
   - Das `IND`-Ereignis von `DigitalInput_I4_CS` löst über `CALIBRATE.CS` den Lesestart aus.
   - Das `IND`-Ereignis von `DigitalInput_I3_CO` löst über `CALIBRATE.CO` den Commit der Kalibrierwerte in die INI-Datei aus (über INI_OFFSET und INI_SCALE).
   - Der kalibrierte Ausgang Y wird an den Splitter AR_SPLIT_2 weitergeleitet.
4. **Signalverteilung**:
   - AR_SPLIT_2.OUT1 leitet den kalibrierten Wert an das Display (Q_NumericValue_PHYSA) zur Anzeige.
   - AR_SPLIT_2.OUT2 leitet den kalibrierten Wert an den Hysterese-Regler (Hysteresis_AR_AX).
5. **Hysterese-Regelung**: Der Hysterese-Regler vergleicht den kalibrierten Eingangswert mit dem Schwellwert (50.5) und der Hysterese (15.3). Der Ausgang wird aktiv, wenn der Wert 50.5 + 15.3/2 erreicht oder überschreitet, und wieder inaktiv, wenn er 50.5 - 15.3/2 unterschreitet; er schaltet den digitalen Ausgang Q2.
6. **Freigabesignal**: Der digitale Eingang I2 gelangt über den Splitter AX_SPLIT_2 an den AnalogInput_I1 (OUT1, als Trigger für die Messung) und gleichzeitig an den Ausgang Q1 (OUT2, direkt weitergeleitet). Somit kann nur bei aktivem I2 der Analogwert eingelesen werden.

Anmerkung: Der Kommentar im Netzwerk weist darauf hin: Ein AD_TO_AR wäre wie ein „reinterpret_cast“ (Bitmuster). AD_TO_AR_NUM wandelt dagegen den Zahlenwert (AD → UDINT → REAL); nur so entsteht ein korrekter Messwert.

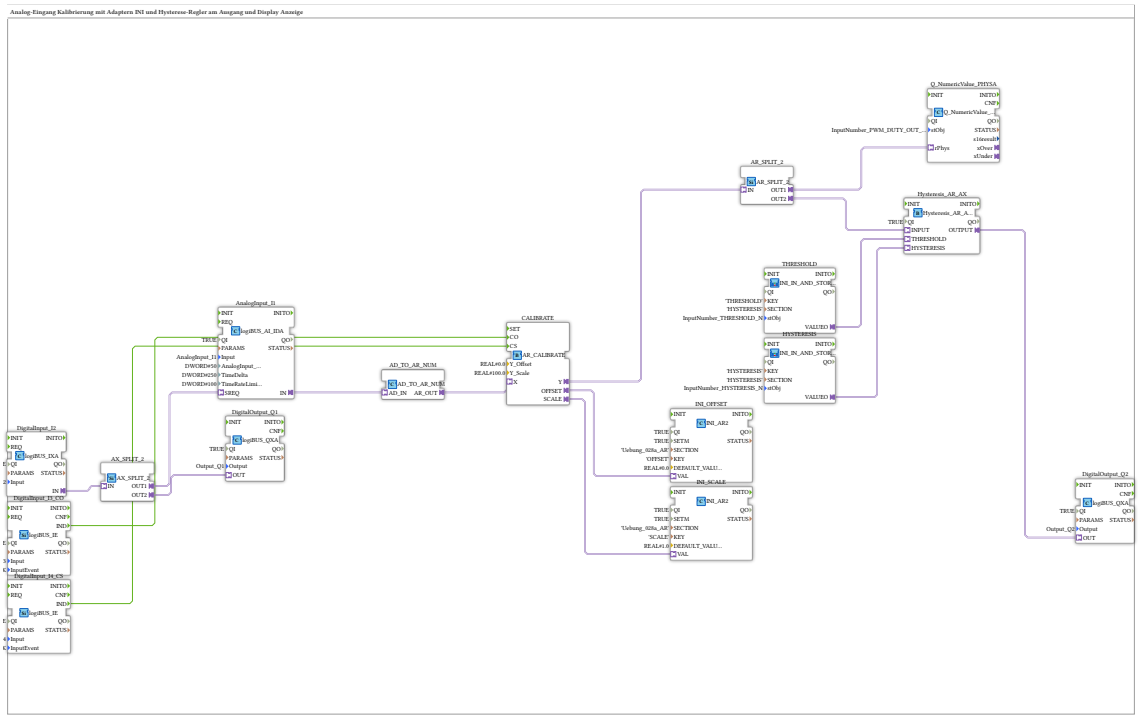
Zusammenfassung

Die Übung „Uebung_028c3_AR“ realisiert eine vollständige Analog-Eingangskalibrierung mit persistenter Speicherung von Offset und Skalierung in einer INI-Datei. Der kalibrierte Wert wird auf einem Display visualisiert und gleichzeitig einem Hysterese-Regler zugeführt, der einen digitalen Ausgang schaltet. Die Steuerung erfolgt über drei digitale Eingänge: Freigabe (I2), Commit (I3) und Start (I4). Die Übung vermittelt den Umgang mit Adaptern, Signalumwandlungen und der Anbindung von INI-Speicherbausteinen in 4diac.

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

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

2.1.2.270 Uebung_028c_AR: Analog-Eingang Kalibrierung mit Adaptern INI und Hysterese-Regler am Ausgang und Display Anzeige



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung kalibriert den Analog-Eingang `I1` mit dem Baustein `AR_CALIBRATE`; die Kalibrierwerte `OFFSET` und `SCALE` werden über zwei `INI_AR2`-Bausteine in einer INI-Datei abgelegt. Der kalibrierte Wert `Y` wird über `AR_SPLIT_2` verteilt: Ein Zweig geht an eine numerische Anzeige (`Q_NumericValue_PHYSA`), der andere an einen Hysterese-Baustein (`Hysteresis_AR_AX`), dessen Ergebnis den Digitalausgang `Q2` schaltet. Schwellwert und Hysterese liefern die beiden SubApps `THRESHOLD` und `HYSTERESIS` (`INI_IN_AND_STORE_AR`). Der Taster `I2` schaltet den Ausgang `Q1` und ist über `AX_SPLIT_2` zugleich mit `SREQ` des Analogeingangs verbunden.

> **Wichtig**: Ein `AD_TO_AR` wäre wie ein „reinterpret_cast“ (Bitmuster). `AD_TO_AR_NUM` wandelt dagegen den Zahlenwert (AD ⟶ UDINT ⟶ REAL).

> **Beispiel** (Kommentar im Netzwerk): `0.0` entspricht 0,5 V, `100.0` entspricht 4,5 V (typischer Automotive-Sensor).

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_I2**: logiBUS Digitaleingang
  - **Typ**: `logiBUS::io::DI::logiBUS_IXA`
  - **Parameter**: `QI = TRUE`, `Input = Input_I2`
  - **Erklärung**: Liefert den Zustand von `I2` als Adapter.
- **AX_SPLIT_2**: Verteilung des Adapters auf zwei Ziele
  - **Typ**: `adapter::events::unidirectional::AX_SPLIT_2`
  - **Parameter**: keine
  - **Erklärung**: `OUT1` geht an `AnalogInput_I1.SREQ`, `OUT2` an `DigitalOutput_Q1`.
- **DigitalOutput_Q1**: logiBUS Digitalausgang
  - **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
  - **Parameter**: `QI = TRUE`, `Output = Output_Q1`
  - **Erklärung**: Gibt `AX_SPLIT_2.OUT2` am physischen Ausgang `Q1` aus.
- **AnalogInput_I1**: logiBUS Analogeingang

  - **Typ**: logiBUS::io::AI::logiBUS_AI_IDA
  - **Parameter**: QI = TRUE, Input = AnalogInput_I1, AnalogInput_hysteresis = 50, TimeDelta = 250, TimeRateLimit = 100
  - **Erklärung**: Liefert den Analogwert von I1 über den Adapter IN.
- **AD_TO_AR_NUM**: numerische Konvertierung
  - **Typ**: adapter::conversion::unidirectional::AD_TO_AR_NUM
  - **Parameter**: keine
  - **Erklärung**: Wandelt AD_IN numerisch (AD → UDINT → REAL) in AR_OUT.
- **DigitalInput_I3_CO**: Taster für das Ereignis CO
  - **Typ**: logiBUS::io::DI::logiBUS_IE
  - **Parameter**: QI = TRUE, Input = Input_I3, InputEvent = BUTTON_SINGLE_CLICK
  - **Erklärung**: Meldet einen Einzelklick auf I3 als IND.
- **DigitalInput_I4_CS**: Taster für das Ereignis CS
  - **Typ**: logiBUS::io::DI::logiBUS_IE
  - **Parameter**: QI = TRUE, Input = Input_I4, InputEvent = BUTTON_SINGLE_CLICK
  - **Erklärung**: Meldet einen Einzelklick auf I4 als IND.
- **CALIBRATE**: Kalibrierung
  - **Typ**: adapter::Engineering::measurements::AR_CALIBRATE
  - **Parameter**: Y_Offset = 0.0, Y_Scale = 100.0
  - **Erklärung**: Erhält den Messwert über X sowie die Ereignisse CO und CS; liefert den kalibrierten Wert Y sowie OFFSET und SCALE.
- **INI_OFFSET**: Ablage von OFFSET in der INI-Datei
  - **Typ**: eclipse4diac::storage::INI_AR2
  - **Parameter**: QI = TRUE, SETM = TRUE, SECTION = 'Uebung_028a_AR', KEY = 'OFFSET', DEFAULT_VALUE = 0.0
  - **Erklärung**: Speichert den Wert von CALIBRATE.OFFSET (Eingang VAL).
- **INI_SCALE**: Ablage von SCALE in der INI-Datei
  - **Typ**: eclipse4diac::storage::INI_AR2
  - **Parameter**: QI = TRUE, SETM = TRUE, SECTION = 'Uebung_028a_AR', KEY = 'SCALE', DEFAULT_VALUE = 1.0
  - **Erklärung**: Speichert den Wert von CALIBRATE.SCALE (Eingang VAL).
- **AR_SPLIT_2**: Verteilung des kalibrierten Werts
  - **Typ**: adapter::events::unidirectional::AR_SPLIT_2
  - **Parameter**: keine
  - **Erklärung**: OUT1 geht an die Anzeige, OUT2 an den Hysterese-Baustein.
- **Q_NumericValue_PHYSA**: numerische Anzeige
  - **Typ**: isobus::UT::Q::Q_NumericValue_PHYSA
  - **Parameter**: stObj = InputNumber_PWM_DUTY_OUT_N
  - **Erklärung**: Zeigt den über rPhys empfangenen kalibrierten Wert an.
- **Hysteresis_AR_AX**: Hysterese-Baustein
  - **Typ**: logiBUS::signalprocessing::hysteresis::Hysteresis_AR_AX
  - **Parameter**: QI = TRUE
  - **Erklärung**: Erhält INPUT, THRESHOLD und HYSTERESIS als REAL-Adapter und liefert OUTPUT als digitalen Adapter.
- **DigitalOutput_Q2**: logiBUS Digitalausgang
  - **Typ**: logiBUS::io::DQ::logiBUS_QXA
  - **Parameter**: QI = TRUE, Output = Output_Q2
  - **Erklärung**: Gibt Hysteresis_AR_AX.OUTPUT am physischen Ausgang Q2 aus.

Sub-Bausteine: THRESHOLD und HYSTERESIS

- **THRESHOLD** und **HYSTERESIS**: Instanzen von MyLib::sys::INI_IN_AND_STORE_AR
  - **Parameter THRESHOLD**: KEY = 'THRESHOLD', SECTION = 'HYSTERESIS', stObj = InputNumber_THRESHOLD_N
  - **Parameter HYSTERESIS**: KEY = 'HYSTERESIS', SECTION = 'HYSTERESIS', stObj = InputNumber_HYSTERESIS_N
  - **Erklärung**: Intern ein Numeric Value_PHYSA (Eingabe), ein INI_AR (Speicherung unter SECTION/KEY) und ein Q_NumericValue_PHYSA (Anzeige); der Wert wird über den Plug VALUEO ausgegeben.

Programmablauf und Verbindungen

1. `DigitalInput_I2.IN` ⟶ `AX_SPLIT_2.IN`; `AX_SPLIT_2.OUT1` ⟶ `AnalogInput_I1.SREQ` und `AX_SPLIT_2.OUT2` ⟶ `DigitalOutput_Q1.OUT` (AdapterConnections).
2. `AnalogInput_I1.IN` ⟶ `AD_TO_AR_NUM.AD_IN` und `AD_TO_AR_NUM.AR_OUT` ⟶ `CALIBRATE.X` (AdapterConnections): Der Analogwert wird numerisch konvertiert.
3. `DigitalInput_I3_CO.IND` ⟶ `CALIBRATE.CO` und `DigitalInput_I4_CS.IND` ⟶ `CALIBRATE.CS` (EventConnections).
4. `CALIBRATE.OFFSET` ⟶ `INI_OFFSET.VAL` und `CALIBRATE.SCALE` ⟶ `INI_SCALE.VAL` (AdapterConnections): Ablage der Kalibrierwerte.
5. `CALIBRATE.Y` ⟶ `AR_SPLIT_2.IN`; `AR_SPLIT_2.OUT1` ⟶ `Q_NumericValue_PHYSA.rPhys` und `AR_SPLIT_2.OUT2` ⟶ `Hysteresis_AR_AX.INPUT` (AdapterConnections): Der kalibrierte Wert geht an die Anzeige und an den Hysterese-Baustein.
6. `THRESHOLD.VALUEO` ⟶ `Hysteresis_AR_AX.THRESHOLD` und `HYSTERESIS.VALUEO` ⟶ `Hysteresis_AR_AX.HYSTERESIS` (AdapterConnections): Schwellwert und Hysterese.
7. `Hysteresis_AR_AX.OUTPUT` ⟶ `DigitalOutput_Q2.OUT` (AdapterConnection): Das Hysterese-Ergebnis schaltet `Q2`.

Zusammenfassung

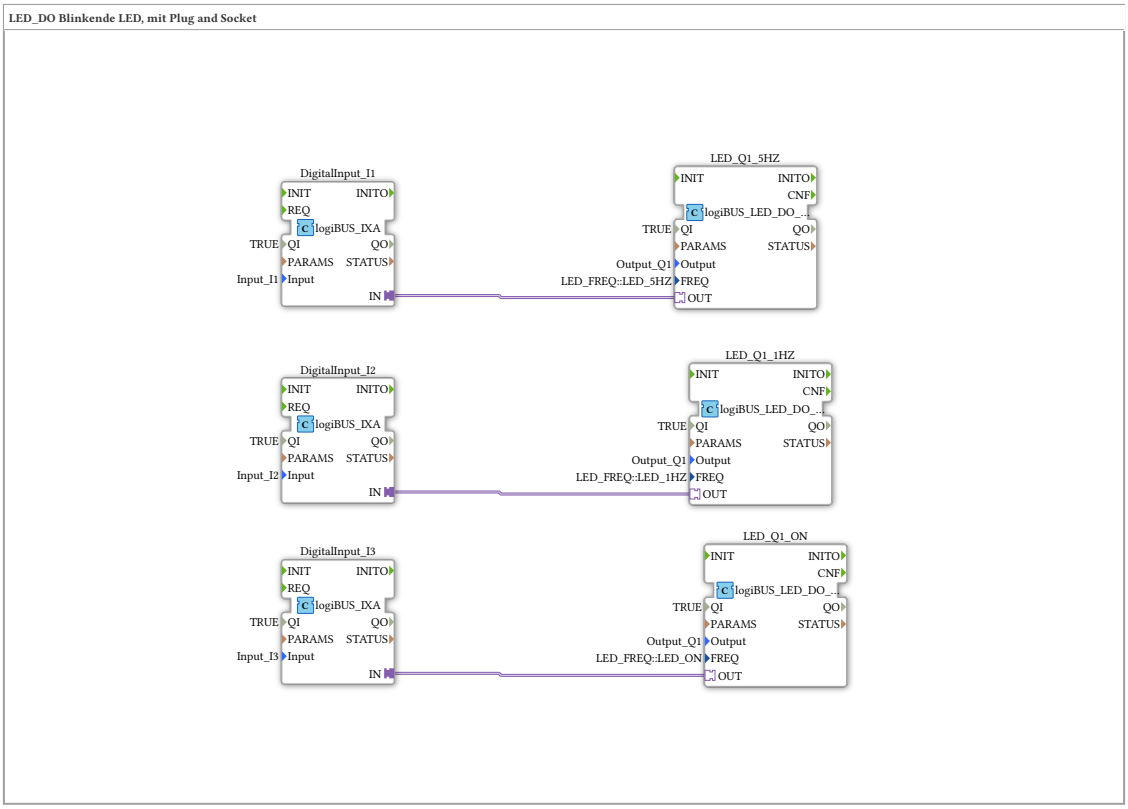
Die Übung 028c_AR erweitert die Kalibrierung aus `Uebung_028a_AX` um eine Anzeige des kalibrierten Werts und um einen Hysterese-Regler, dessen Schwellwert und Hysterese über die SubApps `THRESHOLD` und `HYSTERESIS` eingegeben und in der INI-Sektion `HYSTERESIS` abgelegt werden. Alle Signale laufen durchgehend über Adapter; die numerische Konvertierung mit `AD_TO_AR_NUM` ist dabei wichtig, weil ein einfaches `AD_TO_AR` den Wert nur umdeuten würde.

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

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

2.1.2.271 Uebung_029_AX: LED_DO Blinkende LED, mit Plug and Socket

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_029_AX (LED_DO Blinkende LED, mit Plug and Socket).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **LED_DO Blinkende LED, mit Plug and Socket**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_029_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **LED_Q1_5HZ**: Instanz des Typs logiBUS::io::DO_LED::logiBUS_LED_DO_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
  - Parameter FREQ = LED_FREQ::LED_5HZ
- **DigitalInput_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
- **LED_Q1_1HZ**: Instanz des Typs logiBUS::io::DO_LED::logiBUS_LED_DO_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
  - Parameter FREQ = LED_FREQ::LED_1HZ
- **DigitalInput_I2**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I2
- **LED_Q1_ON**: Instanz des Typs logiBUS::io::DO_LED::logiBUS_LED_DO_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
  - Parameter FREQ = LED_FREQ::LED_ON
- **DigitalInput_I3**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I3

Verbindungen und Schnittstellen

Adapterverbindungen:

- DigitalInput_I1.IN -> LED_Q1_5HZ.OUT
- DigitalInput_I2.IN -> LED_Q1_1HZ.OUT
- DigitalInput_I3.IN -> LED_Q1_ON.OUT

Programmablauf und Funktionsweise

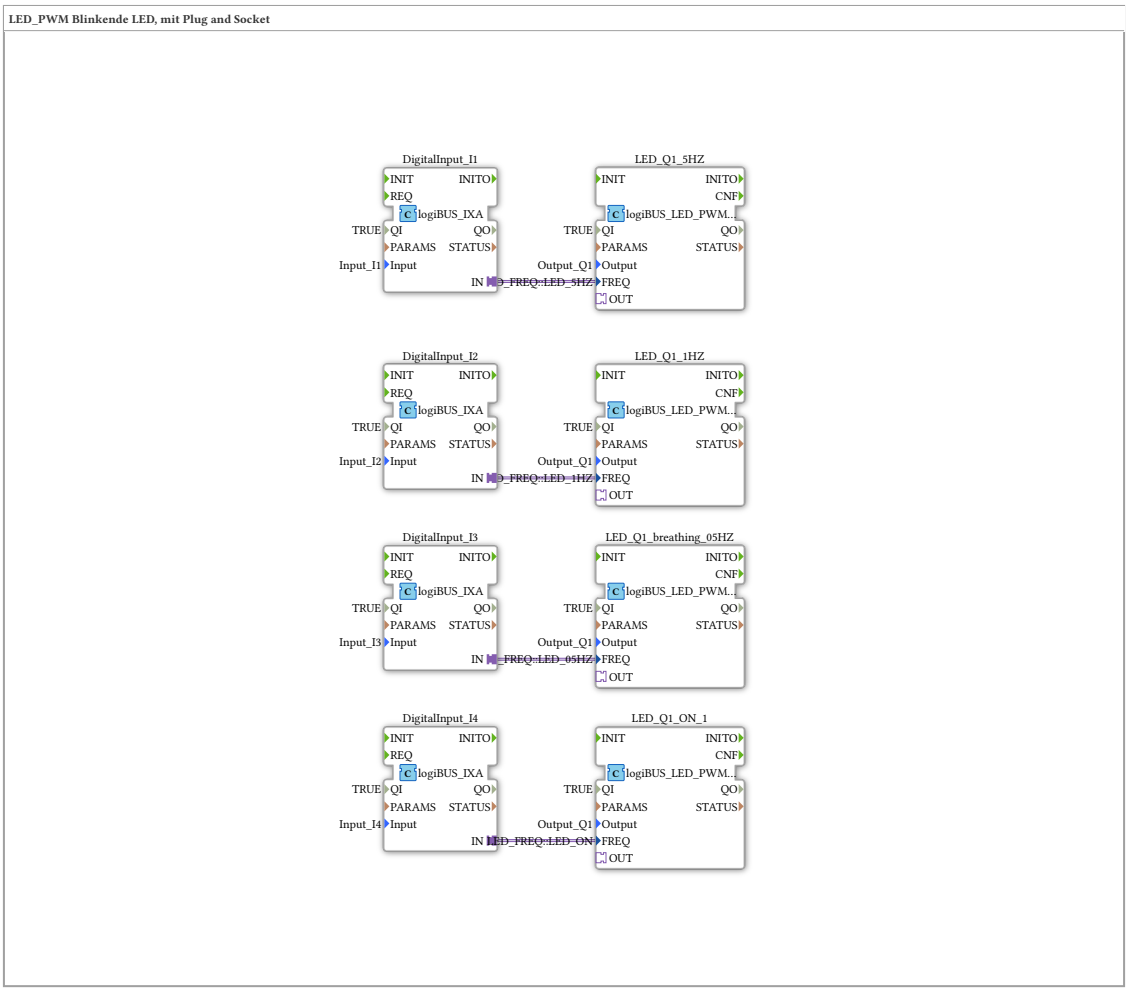
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_029_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.272 Uebung_030_AX: LED_PWM Blinkende LED, mit Plug and Socket

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_030_AX (LED_PWM Blinkende LED, mit Plug and Socket).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **LED_PWM Blinkende LED, mit Plug and Socket**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_030_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **LED_Q1_5HZ**: Instanz des Typs logiBUS::io::DO_LED::logiBUS_LED_PWM_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
  - Parameter FREQ = LED_FREQ::LED_5HZ
- **DigitalInput_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
- **LED_Q1_1HZ**: Instanz des Typs logiBUS::io::DO_LED::logiBUS_LED_PWM_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
  - Parameter FREQ = LED_FREQ::LED_1HZ
- **DigitalInput_I2**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I2
- **LED_Q1_breathing_05HZ**: Instanz des Typs logiBUS::io::DO_LED::logiBUS_LED_PWM_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
  - Parameter FREQ = LED_FREQ::LED_05HZ
- **DigitalInput_I3**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I3
- **DigitalInput_I4**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I4
- **LED_Q1_ON_1**: Instanz des Typs logiBUS::io::DO_LED::logiBUS_LED_PWM_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
  - Parameter FREQ = LED_FREQ::LED_ON

Verbindungen und Schnittstellen

Adapterverbindungen:

- DigitalInput_I1.IN -> LED_Q1_5HZ.OUT
- DigitalInput_I2.IN -> LED_Q1_1HZ.OUT
- DigitalInput_I3.IN -> LED_Q1_breathing_05HZ.OUT
- DigitalInput_I4.IN -> LED_Q1_ON_1.OUT

Programmablauf und Funktionsweise

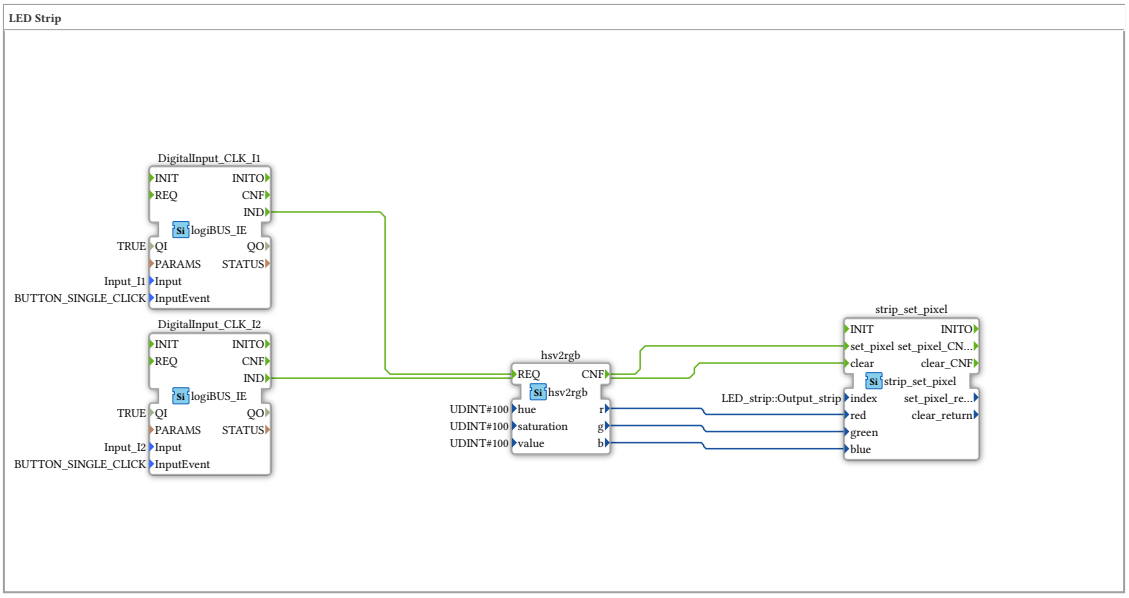
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_030_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.273 Uebung_031_AX: LED Strip

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_031_AX (LED Strip).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **LED Strip**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_031_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_CLK_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **DigitalInput_CLK_I2**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I2
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **strip_set_pixel**: Instanz des Typs logiBUS::esp32::rgb::strip_set_pixel.
  - Parameter index = LED_strip::Output_strip
- **hsv2rgb**: Instanz des Typs logiBUS::esp32::rgb::hsv2rgb.
  - Parameter hue = 100
  - Parameter saturation = 100
  - Parameter value = 100

Verbindungen und Schnittstellen

Ereignisverbindungen:

- DigitalInput_CLK_I2.IND -> strip_set_pixel.clear
- hsv2rgb.CNF -> strip_set_pixel.set_pixel
- DigitalInput_CLK_I1.IND -> hsv2rgb.REQ

Datenverbindungen:

- hsv2rgb.r -> strip_set_pixel.red
- hsv2rgb.g -> strip_set_pixel.green
- hsv2rgb.b -> strip_set_pixel.blue

Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.

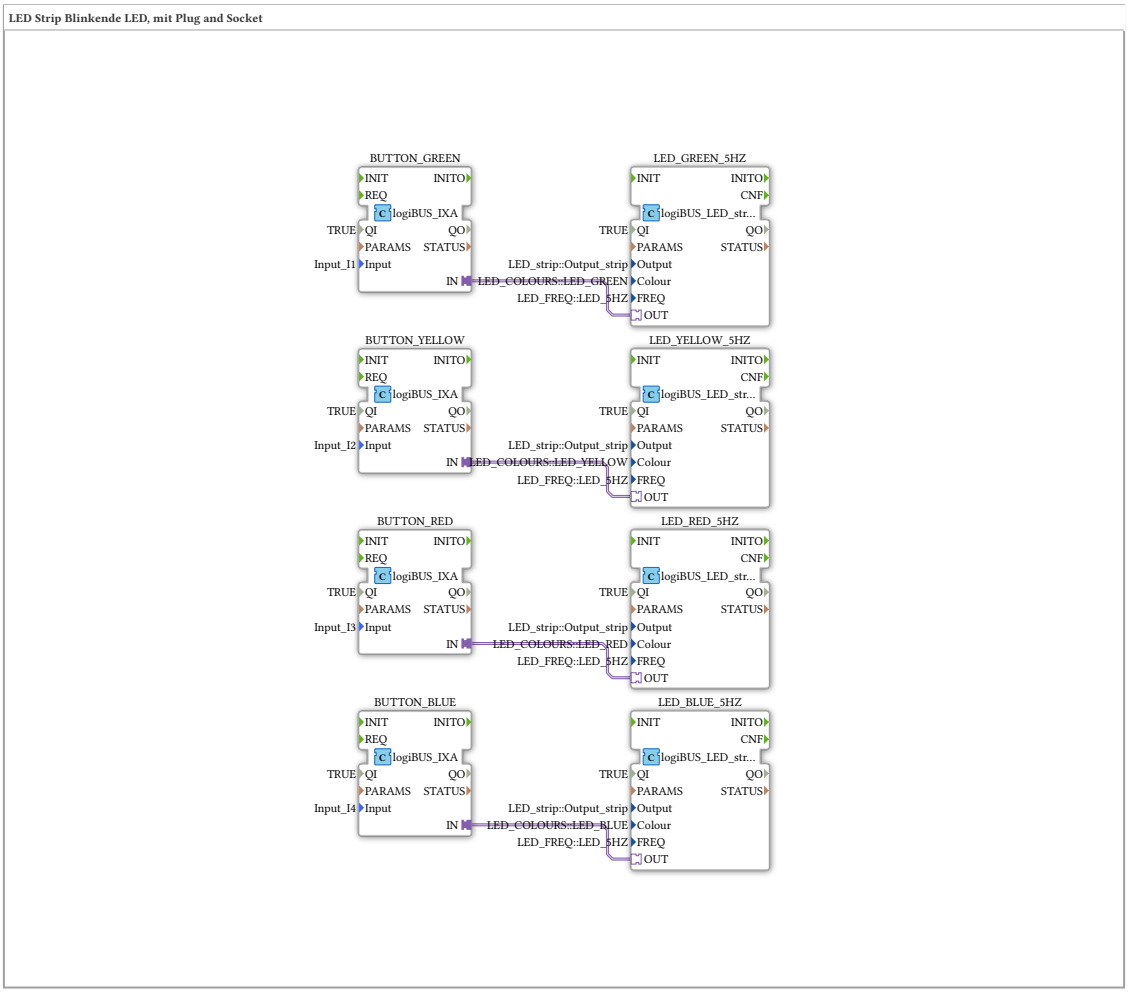
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_031_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.274 Uebung_032_AX: LED Strip Blinkende LED, mit Plug and Socket

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_032_AX (LED Strip Blinkende LED, mit Plug and Socket).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **LED Strip Blinkende LED, mit Plug and Socket**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_032_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **LED_GREEN_5HZ**: Instanz des Typs logiBUS::io::DO_LED::logiBUS_LED_strip_QXA.
  - Parameter QI = TRUE
  - Parameter Output = LED_strip::Output_strip

  - Parameter Colour = LED_COLOURS::LED_GREEN
  - Parameter FREQ = LED_FREQ::LED_5HZ
- **BUTTON_GREEN**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
- **BUTTON_YELLOW**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I2
- **LED_YELLOW_5HZ**: Instanz des Typs logiBUS::io::DO_LED::logiBUS_LED_strip_QXA.
  - Parameter QI = TRUE
  - Parameter Output = LED_strip::Output_strip
  - Parameter Colour = LED_COLOURS::LED_YELLOW
  - Parameter FREQ = LED_FREQ::LED_5HZ
- **BUTTON_RED**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I3
- **LED_RED_5HZ**: Instanz des Typs logiBUS::io::DO_LED::logiBUS_LED_strip_QXA.
  - Parameter QI = TRUE
  - Parameter Output = LED_strip::Output_strip
  - Parameter Colour = LED_COLOURS::LED_RED
  - Parameter FREQ = LED_FREQ::LED_5HZ
- **BUTTON_BLUE**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I4
- **LED_BLUE_5HZ**: Instanz des Typs logiBUS::io::DO_LED::logiBUS_LED_strip_QXA.
  - Parameter QI = TRUE
  - Parameter Output = LED_strip::Output_strip
  - Parameter Colour = LED_COLOURS::LED_BLUE
  - Parameter FREQ = LED_FREQ::LED_5HZ

Verbindungen und Schnittstellen

Adapterverbindungen:

- BUTTON_GREEN.IN -> LED_GREEN_5HZ.OUT
- BUTTON_YELLOW.IN -> LED_YELLOW_5HZ.OUT
- BUTTON_RED.IN -> LED_RED_5HZ.OUT
- BUTTON_BLUE.IN -> LED_BLUE_5HZ.OUT

Programmablauf und Funktionsweise

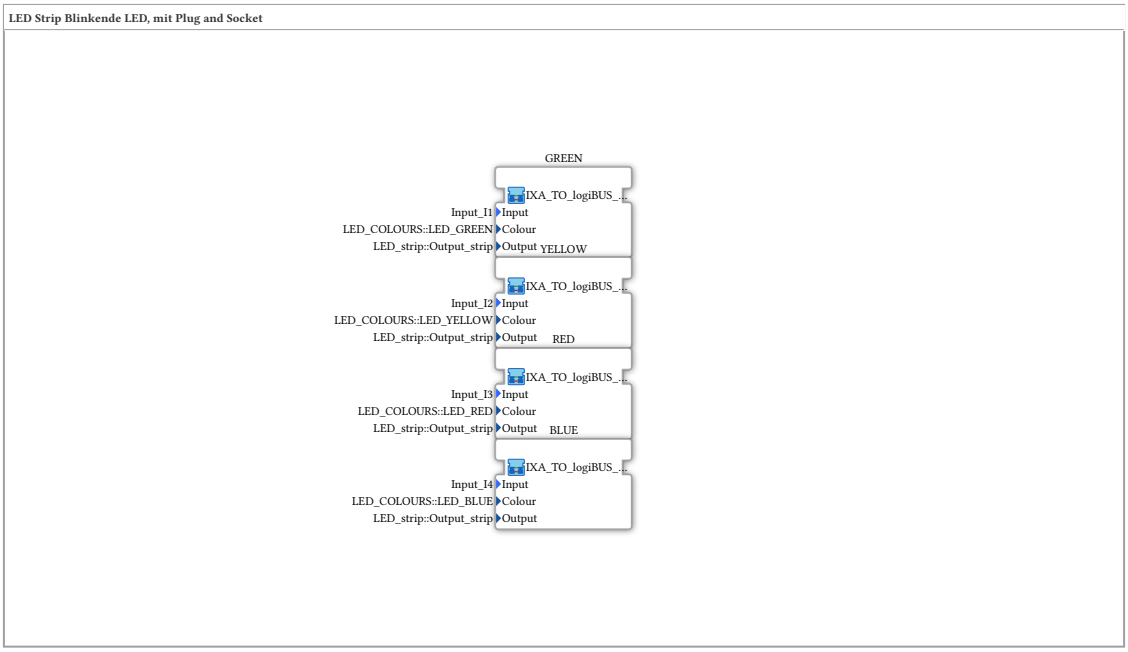
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_032_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.275 Uebung_033_AX: LED Strip Blinkende LED, mit Plug and Socket

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_033_AX (LED Strip Blinkende LED, mit Plug and Socket).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **LED Strip Blinkende LED, mit Plug and Socket**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_033_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Programmablauf und Funktionsweise

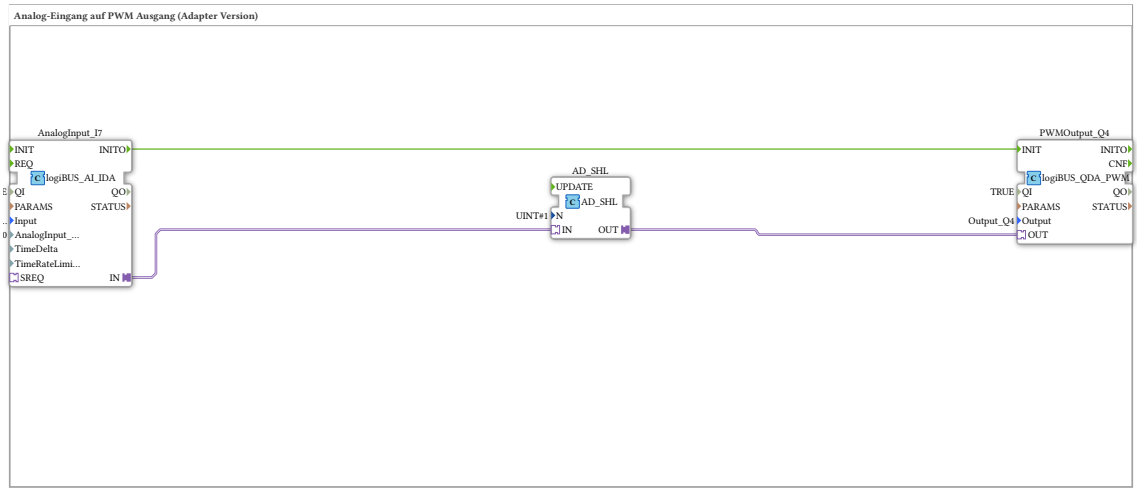
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Signalweiterleitung**: Die Bausteine verarbeiten Daten- und Steuersignale gemäß ihrer Konfiguration.

Zusammenfassung

Die Übung Uebung_033_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.276 Uebung_034_AD: Analog-Eingang auf PWM Ausgang (Adapter Version)

Analog-Eingang auf PWM Ausgang (Adapter Version)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert die Verwendung eines analogen Eingangs zur Ansteuerung eines PWM-Ausgangs über eine Adapterverbindung. Das Signal des analogen Eingangs wird zuerst durch eine Bitverschiebung (Shift Left) verarbeitet, bevor es an den PWM-Ausgang weitergegeben wird. Die Initialisierung des PWM-Ausgangs erfolgt über ein Ereignis, das vom analogen Eingang ausgelöst wird.

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: AnalogInput_I7

- **Typ**: `logiBUS::io::AI::logiBUS_AI_IDA`
- **Verwendete interne FBs**: Keine (Hardwaretreiberbaustein)
  - **Parameter**:
    - `QI` = `TRUE`
    - `Input` = `logiBUS_AI::AnalogInput_I7`
    - `AnalogInput_hysteresis` = `50`
  - **Ereignisausgang/-eingang**:
    - Ereignisausgang `INITO` (wird beim erfolgreichen Initialisieren ausgelöst)
  - **Datenausgang/-eingang**:
    - Adapterausgang `IN` (stellt den gelesenen Analogwert als Adapter bereit)
- **Funktionsweise**: Liest den analogen Eingangswert des angeschlossenen logiBUS-Moduls ein. Der Parameter `AnalogInput_hysteresis` reduziert Signalrauschen. Bei erfolgreicher Initialisierung wird das Ereignis `INITO` gesendet.

Sub-Bausteine: PWMOutput_Q4

- **Typ**: `logiBUS::io::DQ::logiBUS_QDA_PWM`
- **Verwendete interne FBs**: Keine (Hardwaretreiberbaustein)
  - **Parameter**:
    - `QI` = `TRUE`
    - `Output` = `Output_Q4`
  - **Ereignisausgang/-eingang**:
    - Ereigniseingang `INIT` (löst die Initialisierung und Übernahme des PWM-Werts aus)
  - **Datenausgang/-eingang**:
    - Adaptereingang `OUT` (empfängt den Adapter mit dem PWM-Sollwert)
- **Funktionsweise**: Steuert den digitalen Ausgangskanal Q4 als PWM-Ausgang. Der über den Adaptereingang `OUT` erhaltene Wert bestimmt das Tastverhältnis (Pulsweite). Der Ausgang wird mit dem Ereignis `INIT` aktiviert.

Sub-Bausteine: AD_SHL

- **Typ**: `adapter::iec61131::bitwise::AD_SHL`
- **Verwendete interne FBs**: Keine (reine Logik)
  - **Parameter**:
    - `N` = `UINT#1` (Schiebeweite um 1 Bit nach links)
  - **Ereignisausgang/-eingang**: Keine (statische Verarbeitung ohne Ereignisse)
  - **Datenausgang/-eingang**:
    - Adaptereingang `IN` (empfängt den Analogwert als Adapter)
    - Adapterausgang `OUT` (gibt den verschobenen Wert als Adapter aus)
- **Funktionsweise**: Führt eine bitweise Linksverschiebung (Shift Left) um die angegebene Anzahl von Bits (hier 1) auf dem empfangenen Datenwert durch. Dies entspricht einer Multiplikation mit 2. Der verschobene Wert wird über den Adapterausgang bereitgestellt.

Programmablauf und Verbindungen

Der Ablauf beginnt mit der Initialisierung des analogen Eingangs (`AnalogInput_I7`). Sobald dieser erfolgreich initialisiert ist (Ereignis `INITO`), wird das Initialisierungsereignis des PWM-Ausgangs (`PWMOutput_Q4`) über die Ereignisverbindung ausgelöst:

- `AnalogInput_I7.INITO` ⟶ `PWMOutput_Q4.INIT`

Parallel dazu werden die Daten über Adapterverbindungen übertragen:

1. Der Analogwert wird vom Adapterausgang `AnalogInput_I7.IN` an den Adaptereingang `AD_SHL.IN` weitergeleitet.

2. Der aus `AD_SHL.OUT` resultierende verschobene Wert wird an den Adaptereingang `PWMOutput_Q4.OUT` übergeben.

Somit erfolgt die gesamte Datenübertragung adaptierbar und bidirektional über Adapterschnittstellen, ohne separate Datenleitungen. Die Bitverschiebung verstärkt den analogen Eingangswert um den Faktor 2 (entspricht einer Verdopplung), bevor er als PWM-Tastverhältnis ausgegeben wird.

Lernziele:

- Verständnis der Adapterkommunikation in 4diac-IDE
- Einbindung von Analog-Eingängen (logiBUS) und PWM-Ausgängen
- Anwendung bitweiser Operationen (Shift Left) in der Signalverarbeitung
- Verknüpfung von Initialisierungsereignissen zwischen Hardwaretreibern

Schwierigkeitsgrad: Einfach **Benötigte Vorkenntnisse**: Grundlegende Kenntnisse der 4diac-IDE und der logiBUS-Hardware

Zusammenfassung

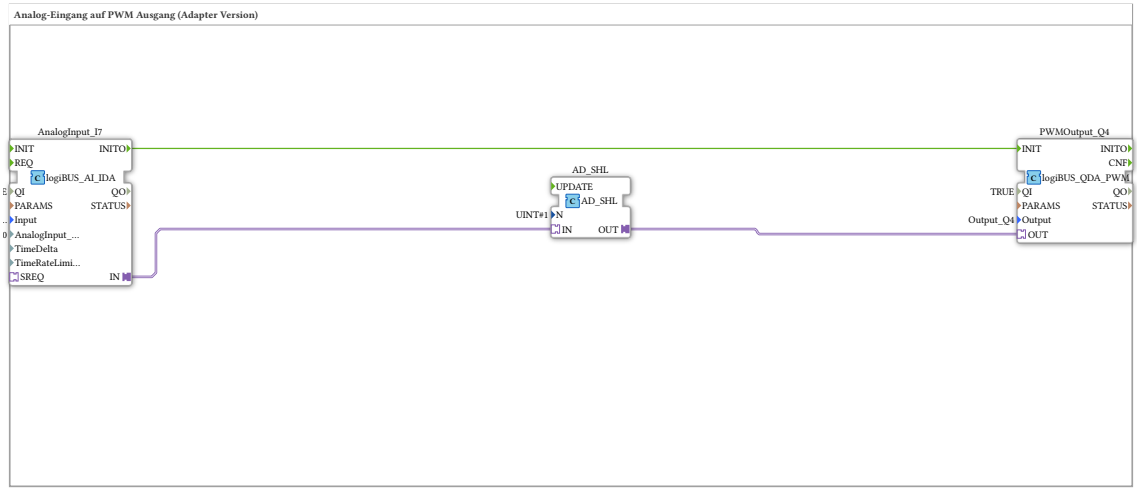
Die Übung „Uebung_034_AD“ zeigt eine einfache, aber praxisrelevante Anwendung: die Umsetzung eines analogen Messwerts in ein PWM-Signal unter Verwendung von Adapterverbindungen. Der Analogwert wird durch eine Linksverschiebung verstärkt und direkt an den PWM-Ausgang weitergeleitet. Die Adaptertechnik ermöglicht eine flexible und typsichere Datenübertragung ohne separate Datenverbindungen. Dieses Beispiel eignet sich für Anwendungen wie Helligkeitssteuerung, Drehzahlregelung oder Signalumformung in der Automatisierungstechnik.

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

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

2.1.2.277 Uebung_034_AX: Analog-Eingang auf PWM Ausgang (Adapter Version)

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_034_AX (Analog-Eingang auf PWM Ausgang (Adapter Version)).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Analog-Eingang auf PWM Ausgang (Adapter Version)**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_034_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **AnalogInput_I7**: Instanz des Typs logiBUS::io::AI::logiBUS_AI_IDA.

  - Parameter QI = TRUE
  - Parameter Input = logiBUS_AI::AnalogInput_I7
  - Parameter AnalogInput_hysteresis = 50
- **PWMOutput_Q4**: Instanz des Typs logiBUS::io::DQ::logiBUS_QDA_PWM.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q4
- **AD_SHL**: Instanz des Typs adapter::iec61131::bitwise::AD_SHL.
  - Parameter N = UINT#1

Verbindungen und Schnittstellen

Adapterverbindungen:

- AnalogInput_I7.IN -> AD_SHL.IN
- AD_SHL.OUT -> PWMOutput_Q4.OUT

Ereignisverbindungen:

- AnalogInput_I7.INITO -> PWMOutput_Q4.INIT

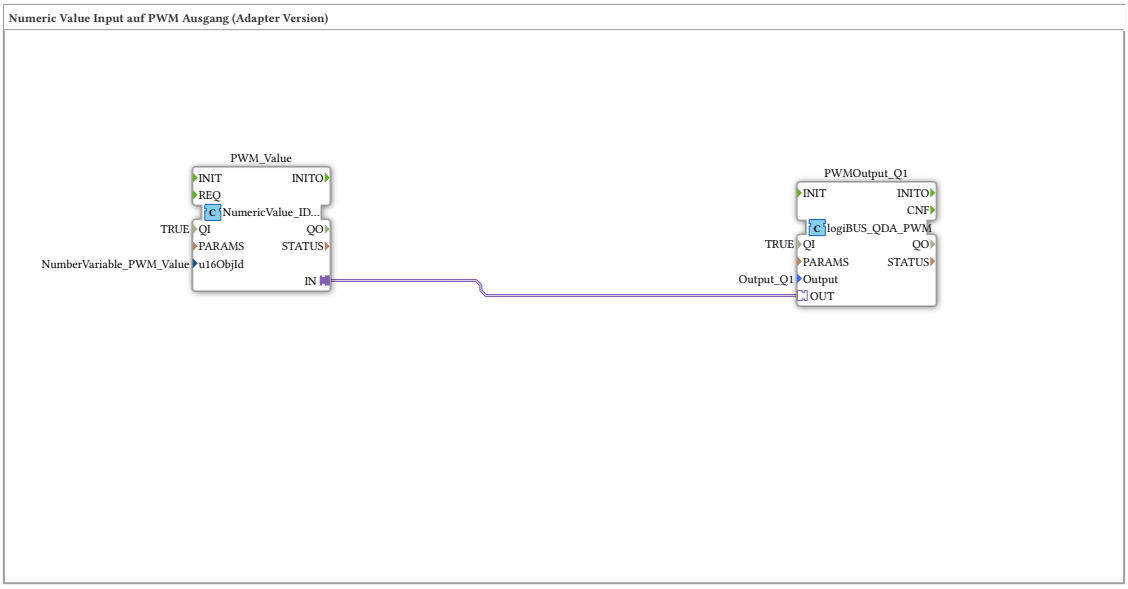
Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_034_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.278 Uebung_034a1_Q1_AX: Numeric Value Input auf PWM Ausgang (Adapter Version)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert die einfache Verbindung einer **Numerischen Werteingabe** (ISOBUS-NumericValue) mit einem **PWM-Ausgang** über eine direkte Adapterverbindung. Der eingegebene Zahlenwert wird nach Bestätigung der Eingabe (OK-Taste) auf den PWM-Ausgang des logiBUS-Moduls (Kanal Q1) ausgegeben.

Die Übung ist als **SubAppType** realisiert und verwendet ausschließlich Adapter für die Signalübergabe, wodurch auf separate Daten- und Ereignisverbindungen verzichtet wird.

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: `Uebung_034a1_Q1_AX`

- **Typ**: SubAppType (zusammengesetzter Baustein)
- **Verwendete interne FBs**:
  - **`PWM_Value`**: `isobus::UT::io::NumericValue::NumericValue_IDA`
    - Parameter:
      - `QI` = `TRUE` (Baustein aktiviert)
      - `u16ObjId` = `InputNumber_PWM_Value` (ISOBUS-Objekt-ID für die numerische Eingabe)
    - **Funktionsweise**: Liest einen vom Benutzer eingegebenen Zahlenwert von einem ISOBUS-Objekt (z. B. Terminal) und stellt diesen über den Adapterausgang `OUT` bereit. Die Daten werden erst nach Drücken der OK-Taste am Terminal aktualisiert.
  - **`PWMOutput_Q1`**: `logiBUS::io::DQ::logiBUS_QDA_PWM`
    - Parameter:
      - `QI` = `TRUE` (Baustein aktiviert)
      - `Output` = `Output_Q1` (logiBUS-Ausgangskanal Q1)
    - **Funktionsweise**: Wandelt den über den Adaptereingang `IN` erhaltenen numerischen Wert in ein PWM-Signal auf dem angegebenen logiBUS-Ausgang (`Output_Q1`) um. Der Wert bestimmt das Tastverhältnis der PWM.

Programmablauf und Verbindungen

1. Der Benutzer gibt an einem ISOBUS-Terminal einen Zahlenwert über das Objekt `InputNumber_PWM_Value` ein.
2. Nach Betätigen der OK-Taste wird der Wert vom FB `PWM_Value` übernommen und an seinem Adapterausgang `IN` bereitgestellt.
3. Der Adapterausgang ist direkt mit dem Adaptereingang `OUT` des FB `PWMOutput_Q1` verbunden:

   ```
   Verbindung: PWM_Value.IN → PWMOutput_Q1.OUT
   ```

4. `PWMOutput_Q1` setzt den empfangenen Wert als PWM-Tastverhältnis auf dem logiBUS-Ausgang `Output_Q1` um.

Hinweis: Wie im Kommentar im Netzwerk vermerkt, wird der aktualisierte Wert **nicht** bei jeder Tastatureingabe (z. B. beim Drehen eines Encoders) übertragen, sondern erst nach drücken der OK-Taste. Dieses Verhalten ist durch den FB `NumericValue_IDA` vorgegeben.

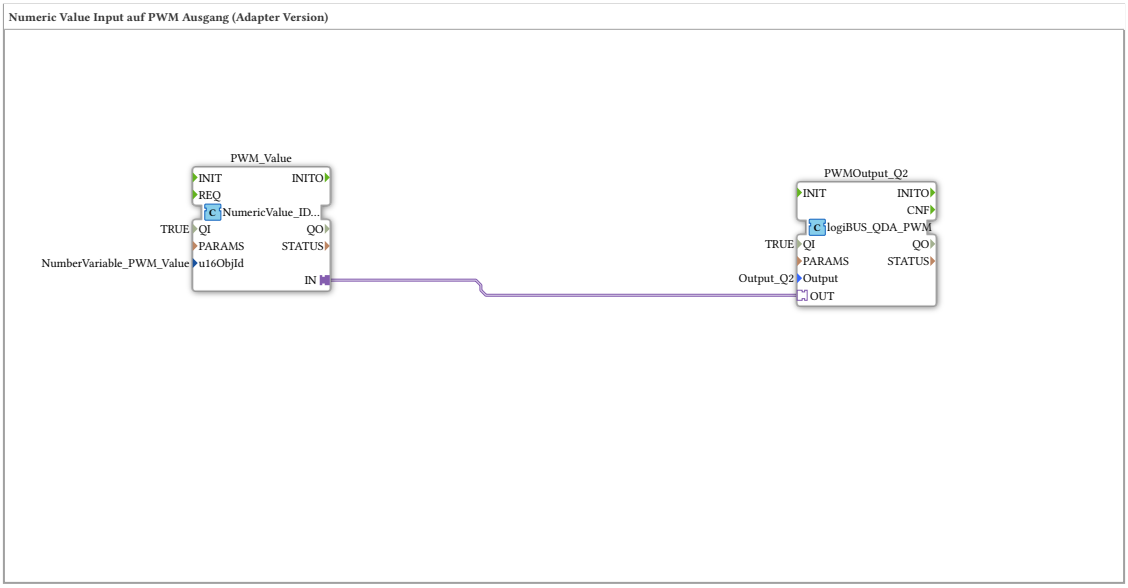
Zusammenfassung

Die Übung zeigt eine **minimale Konfiguration zur Ansteuerung eines PWM-Ausgangs** mittels einer eingegebenen Zahl. Sie verdeutlicht die Verwendung von **Adapterverbindungen** zur Datenübertragung zwischen ISOBUS-Eingabe und Aktor. Der Aufbau ist einfach, erfordert jedoch das Verständnis der ISOBUS-Kommunikation und der PWM-Parametrierung. Die Übung eignet sich für Einsteiger in die 4diac-IDE und die logiBUS-Hardware.

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

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

2.1.2.279 Uebung_034a1_Q2_AX: Numeric Value Input auf PWM Ausgang (Adapter Version)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert die Kopplung eines numerischen Eingabewerts (über einen iSoBUS-Numeric-Value-Dienst) mit einem PWM-Ausgang (logiBUS). Der vom Anwender eingegebene Zahlenwert wird direkt in ein PWM-Signal umgesetzt und am Ausgang `Output_Q2` ausgegeben. Die Kommunikation zwischen den beiden Funktionsbausteinen erfolgt über eine Adapterverbindung, was eine modulare und flexible Verschaltung ermöglicht.

Verwendete Funktionsbausteine (FBs)

Im Netzwerk der Subapplikation werden zwei Funktionsbausteine eingesetzt:

- **PWM_Value** *Typ*: `isobus::UT::io::NumericValue::NumericValue_IDA` *Aufgabe*: Empfängt einen numerischen Wert aus dem iSoBUS-Netzwerk über die Objekt-ID `InputNumber_PWM_Value`. Die Daten stehen nach Bestätigung durch den Benutzer (z. B. Drücken der OK-Taste) am Ausgang zur Verfügung. *Parameter*:
  - `QI` (Einschaltsignal) = `TRUE` (dauerhaft aktiv)
  - `u16ObjId` = `InputNumber_PWM_Value` (iSoBUS-Objekt-ID des Eingabewerts)
- **PWMOutput_Q2** *Typ*: `logiBUS::io::DQ::logiBUS_QDA_PWM` *Aufgabe*: Wandelt den eingehenden Zahlenwert in ein PWM-Signal um und gibt es über den logiBUS-Ausgang `Output_Q2` aus. *Parameter*:
  - `QI` (Einschaltsignal) = `TRUE` (dauerhaft aktiv)
  - `Output` = `Output_Q2` (logiBUS-Ausgangsadresse)

Wichtiger Hinweis: Das Ereignis zur Datenübergabe wird erst ausgelöst, wenn der eingegebene Zahlenwert mit der OK-Taste bestätigt wird – nicht bereits bei jeder Tastenbetätigung. Siehe Kommentar im Netzwerk:

> ACHTUNG !! Ereignis erscheint erst im Adapter, wenn man die Numeric Value mit OK bestätigt.
> nicht bei Button Druck.

Programmablauf und Verbindungen

1. Der Funktionsbaustein `PWM_Value` wartet auf einen gültigen Zahlenwert vom iSoBUS-Eingabefeld.
2. Nach Benutzerbestätigung (OK-Taste) wird über den **Adapterausgang** `IN` ein Datenevent mit dem Zahlenwert gesendet.

3. Die Adapterverbindung leitet dieses Signal an den **Adaptereingang** `OUT` des Funktionsbausteins `PWMOutput_Q2` weiter.
4. `PWMOutput_Q2` setzt den empfangenen Wert in ein PWM-Signal mit entsprechender Pulsweite um und steuert den angeschlossenen logiBUS-Ausgang `Output_Q2`.

Wichtige Vorbedingungen:

- Die beiden Funktionsbausteine sind über ihre Adapterschnittstellen starr verbunden (keine dynamischen Verbindungen).
- Das zugrundeliegende iSoBUS-Objekt (`InputNumber_PWM_Value`) muss im System konfiguriert sein.
- Das logiBUS-Ausgangsmodul (`Output_Q2`) muss vorhanden und adressiert sein.

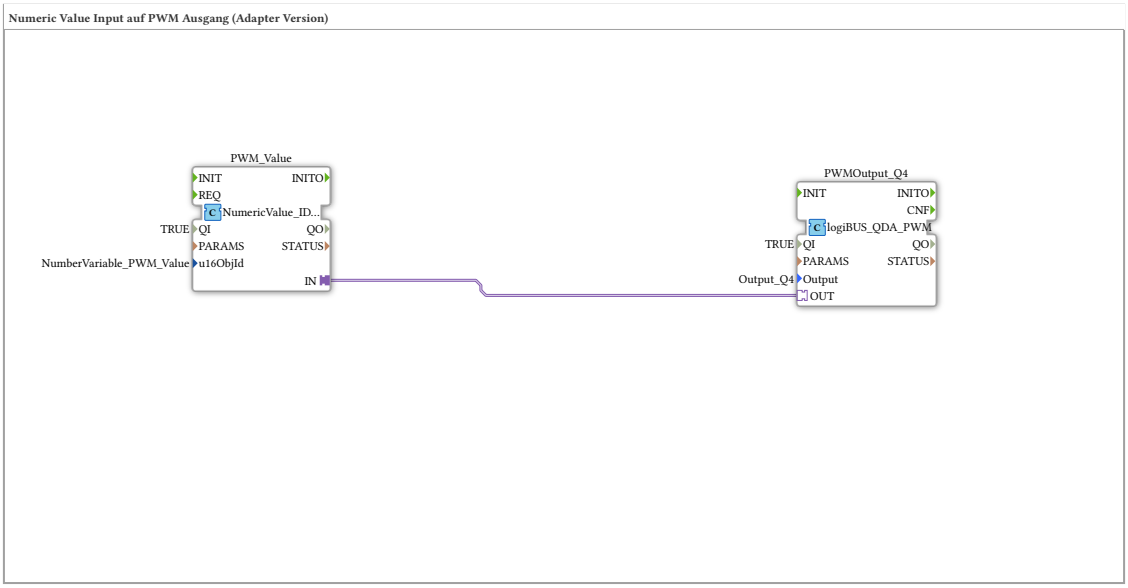
Zusammenfassung

Die Übung zeigt, wie eine numerische Benutzereingabe aus einem iSoBUS-Dienst über eine Adapterverbindung direkt in ein PWM-Ausgangssignal umgesetzt wird. Die Verwendung von Adaptern vereinfacht die Verschaltung und erhöht die Wiederverwendbarkeit der Bausteine. Nach der Übung ist der Lernende in der Lage, iSoBUS-Eingabefelder mit logiBUS-PWM-Ausgängen zu kombinieren und die Besonderheit der Ereignisauslösung (Bestätigung statt Tastendruck) zu berücksichtigen.

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

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

2.1.2.280 Uebung_034a1_Q4_AX: Numeric Value Input auf PWM Ausgang (Adapter Version)



• — ○ • — ○ • — ○ •

Einleitung

Diese Übung demonstriert die Steuerung eines PWM-Ausgangs (logiBUS Output Q4) durch einen numerischen Eingabewert. Die Kommunikation zwischen dem Eingabebaustein und dem Ausgabebaustein erfolgt über Adapter-Verbindungen („Adapter Version“). Ein integrierter Kommentar weist darauf hin, dass das Ereignis für die Wertübernahme erst ausgelöst wird, wenn der eingegebene numerische Wert mit „OK“ bestätigt wird – nicht bereits bei einem Tastendruck.

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: `PWM_Value`

- **Typ**: `isobus::UT::io::NumericValue::NumericValue_IDA`
- **Verwendete interne FBs**: (Keine weiteren internen FBs, da es sich um einen atomaren FB handelt. Der FB selbst ist Teil der Bibliothek `isobus`.)

- **Parameter**:
  - `QI` = `TRUE`
  - `u16ObjId` = `InputNumber_PWM_Value` (verweist auf eine im Projekt definierte Numeric-Value-Instanz)
- **Funktionsweise**: Der FB liest den vom Benutzer eingegebenen numerischen Wert (z. B. aus einem HMI-Eingabefeld) und stellt diesen über seine Adapter-Schnittstelle (`IN`) bereit. Die Ausgabe erfolgt erst, nachdem die Eingabe mit „OK“ bestätigt wurde. Die Ereignissteuerung wird implizit über den Adapter abgewickelt.

Sub-Bausteine: `PWMOutput_Q4`

- **Typ**: `logiBUS::io::DQ::logiBUS_QDA_PWM`
- **Verwendete interne FBs**: (Keine weiteren internen FBs, atomarer FB aus logiBUS-Bibliothek)
- **Parameter**:
  - `QI` = `TRUE`
  - `Output` = `Output_Q4` (logischer Name des physikalischen PWM-Ausgangs am logiBUS-Modul)
- **Funktionsweise**: Der FB empfängt über seinen Adapter-Eingang (`OUT`) den aktuellen Sollwert (z. B. eine Zahl von 0 bis 1000 o. ä.) und stellt diesen als PWM-Signal am spezifizierten Kanal `Output_Q4` zur Verfügung. Der Wert entspricht dem Tastverhältnis der PWM.

Programmablauf und Verbindungen

Der Netzwerkablauf besteht aus zwei Funktionsbausteinen, die ausschließlich über eine Adapter-Verbindung kommunizieren:

- **Quelle**: `PWM_Value.IN` (Ausgangsseite des Adapters)
- **Ziel**: `PWMOutput_Q4.OUT` (Eingangsseite des Adapters)

Die Adapter-Verbindung überträgt den numerischen Wert einschließlich der zugehörigen Ereignissteuerung. Sobald der Benutzer im HMI den Wert bestätigt, wird das Ereignis über den Adapter an den PWM-Ausgangsbaustein weitergeleitet, der daraufhin das PWM-Signal aktualisiert.

Hinweis (aus dem Kommentar im Netzwerk): Das Ereignis wird erst dann über den Adapter gesendet, wenn die numerische Eingabe mit „OK“ quittiert wird – nicht bereits bei einem Tastendruck oder einer Änderung des Eingabefeldes. Dies ist bei der Planung der Bedienoberfläche zu berücksichtigen.

Verbindung	Von	Nach
Adapter	`PWM_Value.IN`	`PWMOutput_Q4.OUT`

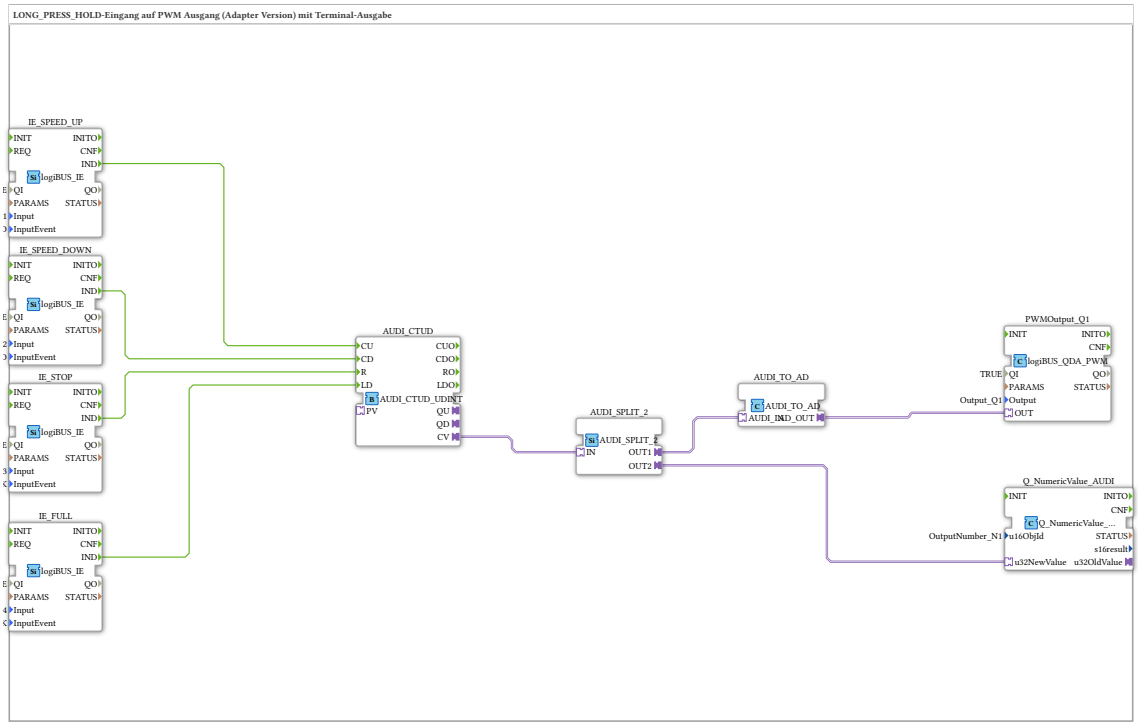
Zusammenfassung

Die Übung `Uebung_034a1_Q4_AX` verbindet einen numerischen Eingabe-FB (`NumericValue_IDA`) mit einem PWM-Ausgangs-FB (`logiBUS_QDA_PWM`) über Adapter. Dadurch wird ein einfaches Zusammenspiel von Bedieneingabe und Hardware-Ausgabe realisiert. Die Besonderheit liegt in der ereignisgesteuerten Übernahme des Wertes erst nach Bestätigung, was eine saubere Trennung von Eingabeänderung und Aktualisierung des Ausgangs ermöglicht.

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

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

2.1.2.281 Uebung_034b_AUDI: LONG_PRESS_HOLD-Eingang auf PWM Ausgang (Adapter Version) mit Terminal-Ausgabe



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert die Verwendung von **LONG_PRESS_HOLD**-Eingängen zur Steuerung eines PWM-Ausgangs. Über vier Taster kann der PWM-Wert erhöht, verringert, zurückgesetzt oder auf einen Maximalwert gesetzt werden. Der aktuelle Zählerstand wird zusätzlich auf einem Terminal ausgegeben. Die gesamte Schaltung ist als **SubApp** mit Adapter-basierter Datenwandlung realisiert.

Verwendete Funktionsbausteine (FBs)

Die SubApp enthält folgende Funktionsbausteine:

Sub-Baustein: IE_SPEED_UP

- **Typ**: `logiBUS::io::DI::logiBUS_IE`
- **Parameter**:
  - `QI` = TRUE
  - `Input` = Input_I1 (physischer Eingang)
  - `InputEvent` = `BUTTON_LONG_PRESS_HOLD`
- **Ereignisausgang**: `IND` (wird bei Langdruck ausgelöst)
- **Funktionsweise**: Erfasst ein Long-Press-Hold-Ereignis am ersten digitalen Eingang und gibt ein IND-Ereignis aus, um den Zähler zu erhöhen.

Sub-Baustein: IE_SPEED_DOWN

- **Typ**: `logiBUS::io::DI::logiBUS_IE`
- **Parameter**:
  - `QI` = TRUE
  - `Input` = Input_I2
  - `InputEvent` = `BUTTON_LONG_PRESS_HOLD`
- **Ereignisausgang**: `IND`
- **Funktionsweise**: Erfasst Long-Press-Hold am zweiten Eingang und löst das Herunterzählen aus.

Sub-Baustein: IE_STOP

- **Typ**: `logiBUS::io::DI::logiBUS_IE`

- **Parameter**:
  - `QI` = TRUE
  - `Input` = Input_I3
  - `InputEvent` = `BUTTON_SINGLE_CLICK`
- **Ereignisausgang**: `IND`
- **Funktionsweise**: Erfasst einen Single-Click am dritten Eingang und setzt den Zähler zurück (Reset).

Sub-Baustein: IE_FULL

- **Typ**: `logiBUS::io::DI::logiBUS_IE`
- **Parameter**:
  - `QI` = TRUE
  - `Input` = Input_I4
  - `InputEvent` = `BUTTON_SINGLE_CLICK`
- **Ereignisausgang**: `IND`
- **Funktionsweise**: Erfasst einen Single-Click am vierten Eingang und lädt einen voreingestellten Maximalwert in den Zähler (Load).

Sub-Baustein: AUDI_CTUD

- **Typ**: `adapter::events::unidirectional::AUDI_CTUD_UDINT` (Aufwärts/Abwärtszähler)
- **Parameter**: Keine (Standardkonfiguration)
- **Ereigniseingänge**:
  - `CU` (Count Up)
  - `CD` (Count Down)
  - `R` (Reset)
  - `LD` (Load)
- **Datenausgang**: `CV` (aktueller Zählerwert, Typ UDINT)
- **Funktionsweise**: Zählt bei jedem `CU`-Ereignis hoch und bei jedem `CD`-Ereignis herunter. Bei `R` wird der Zähler auf 0 gesetzt, bei `LD` auf einen internen Standardwert (z. B. 10000).

Sub-Baustein: AUDI_SPLIT_2

- **Typ**: `adapter::events::unidirectional::AUDI_SPLIT_2` (Signalverteiler)
- **Parameter**: Keine
- **Adaptereingang**: `IN` (Daten)
- **Adapterausgänge**: `OUT1`, `OUT2` (gleicher Wert auf beiden Ausgängen)
- **Funktionsweise**: Verteilt den eingehenden Zählerwert auf zwei parallele Pfade – einen für die PWM-Wandlung und einen für die Terminalausgabe.

Sub-Baustein: AUDI_TO_AD

- **Typ**: `adapter::conversion::unidirectional::AUDI_TO_AD` (Wandler)
- **Parameter**: Keine
- **Adaptereingang**: `AUDI_IN`
- **Adapterausgang**: `AD_OUT` (analoger Wert, z. B. 0–10000)
- **Funktionsweise**: Wandelt den Zählerwert (AUDI-Format) in einen analogen Wert um, der für die PWM-Stufe geeignet ist.

Sub-Baustein: PWMOutput_Q1

- **Typ**: `logiBUS::io::DQ::logiBUS_QDA_PWM`
- **Parameter**:
  - `QI` = TRUE (aktiviert)
  - `Output` = `Output_Q1` (physischer Ausgang)
- **Dateneingang**: `OUT` (vom Wandler)
- **Funktionsweise**: Gibt ein PWM-Signal proportional zum analogen Eingangswert am Ausgang `Q1` aus.

Sub-Baustein: Q_NumericValue_AUDI

- **Typ**: `isobus::UT::Q::Q_NumericValue_AUDI`
- **Parameter**:
  - `u16ObjId` = `OutputNumber_N1` (Zieladresse für das Terminal)
- **Dateneingang**: `u32NewValue` (aktueller Zählerwert)

- **Funktionsweise**: Sendet den Zählerwert an die konfigurierte Ausgabenummer des Terminals zur numerischen Anzeige.

Programmablauf und Verbindungen

Die Ereignis- und Datenflüsse sind wie folgt verknüpft:

- **Ereignisverbindungen** (von den `IE`-Bausteinen zum Zähler):
  - `IE_SPEED_UP.IND` ⟶ `AUDI_CTUD.CU` (hochzählen)
  - `IE_SPEED_DOWN.IND` ⟶ `AUDI_CTUD.CD` (herunterzählen)
  - `IE_STOP.IND` ⟶ `AUDI_CTUD.R` (Reset)
  - `IE_FULL.IND` ⟶ `AUDI_CTUD.LD` (Maximalwert laden)
- **Datenverbindungen** (Adapter):
  - `AUDI_CTUD.CV` ⟶ `AUDI_SPLIT_2.IN`
  - `AUDI_SPLIT_2.OUT1` ⟶ `AUDI_TO_AD.AUDI_IN`
  - `AUDI_TO_AD.AD_OUT` ⟶ `PWMOutput_Q1.OUT`
  - `AUDI_SPLIT_2.OUT2` ⟶ `Q_NumericValue_AUDI.u32NewValue`

Ablauf: Durch Long-Press-Hold an den Tastern für Geschwindigkeit hoch/runter wird der Aufwärts-/Abwärtszähler inkrementiert bzw. dekrementiert. Ein Single-Click auf *Stop* setzt den Zähler zurück, *Full* lädt einen vollen Wert. Der Zählerstand wird über einen Splitter zum einen in ein PWM-Signal umgewandelt und ausgegeben, zum anderen auf dem Terminal angezeigt.

Lernziele:

- Verständnis für die Kopplung von Ereignis- und Datenflüssen in 4diac
- Nutzung von Long-Press-Hold- und Single-Click-Ereignissen
- Adapterbasierte Datenwandlung und Signalverteilung
- Einbindung von Terminalausgaben zur Wertanzeige

Schwierigkeitsgrad: Mittel **Vorkenntnisse**: Grundlegende Bedienung der 4diac-IDE, Umgang mit logiBUS-Funktionsbausteinen

Zusammenfassung

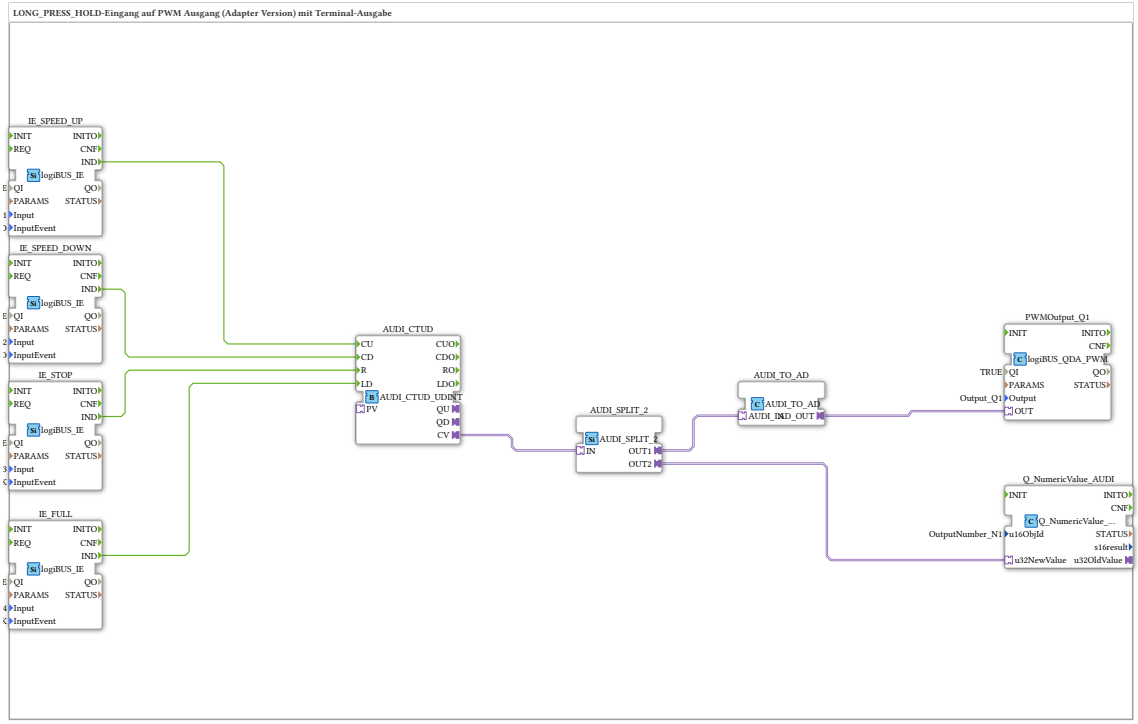
Die Übung **Uebung_034b_AUDI** realisiert eine Geschwindigkeitssteuerung über vier Taster. Long-Press-Hold erhöht oder verringert das PWM-Signal, Single-Click setzt zurück oder auf Maximum. Die Architektur verwendet Adapter zur Aufteilung und Wandlung des Zählerwerts. Die Terminalausgabe ermöglicht eine einfache Überwachung des aktuellen Werts – ideal für das Erlernen von Adapter-basierten Datenflüssen und der Kombination von Ereignis- und Datenverarbeitung.

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

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

2.1.2.282 Uebung_034b_AX: LONG_PRESS_HOLD-Eingang auf PWM Ausgang (Adapter Version) mit Terminal-Ausgabe

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_034b_AX (LONG_PRESS_HOLD-Eingang auf PWM Ausgang (Adapter Version) mit Terminal-Ausgabe).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **LONG_PRESS_HOLD-Eingang auf PWM Ausgang (Adapter Version) mit Terminal-Ausgabe**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_034b_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **PWMOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QDA_PWM.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
- **IE_SPEED_UP**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
  - Parameter InputEvent = BUTTON_LONG_PRESS_HOLD
- **IE_SPEED_DOWN**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I2
  - Parameter InputEvent = BUTTON_LONG_PRESS_HOLD
- **IE_STOP**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I3
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **IE_FULL**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I4
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **AUDI_CTUD**: Instanz des Typs adapter::events::unidirectional::AUDI_CTUD_UDINT.
- **Q_NumericValue_AUDI**: Instanz des Typs isobus::UT::Q::Q_NumericValue_AUDI.
  - Parameter u16ObjId = OutputNumber_N1
- **AUDI_SPLIT_2**: Instanz des Typs adapter::events::unidirectional::AUDI_SPLIT_2.
- **AUDI_TO_AD**: Instanz des Typs adapter::conversion::unidirectional::AUDI_TO_AD.

Verbindungen und Schnittstellen

Adapterverbindungen:

- AUDI_CTUD.CV -> AUDI_SPLIT_2.IN
- AUDI_SPLIT_2.OUT2 -> Q_NumericValue_AUDI.u32NewValue
- AUDI_SPLIT_2.OUT1 -> AUDI_TO_AD.AUDI_IN
- AUDI_TO_AD.AD_OUT -> PWMOutput_Q1.OUT

Ereignisverbindungen:

- IE_SPEED_DOWN.IND -> AUDI_CTUD.CD
- IE_STOP.IND -> AUDI_CTUD.R
- IE_FULL.IND -> AUDI_CTUD.LD
- IE_SPEED_UP.IND -> AUDI_CTUD.CU

Programmablauf und Funktionsweise

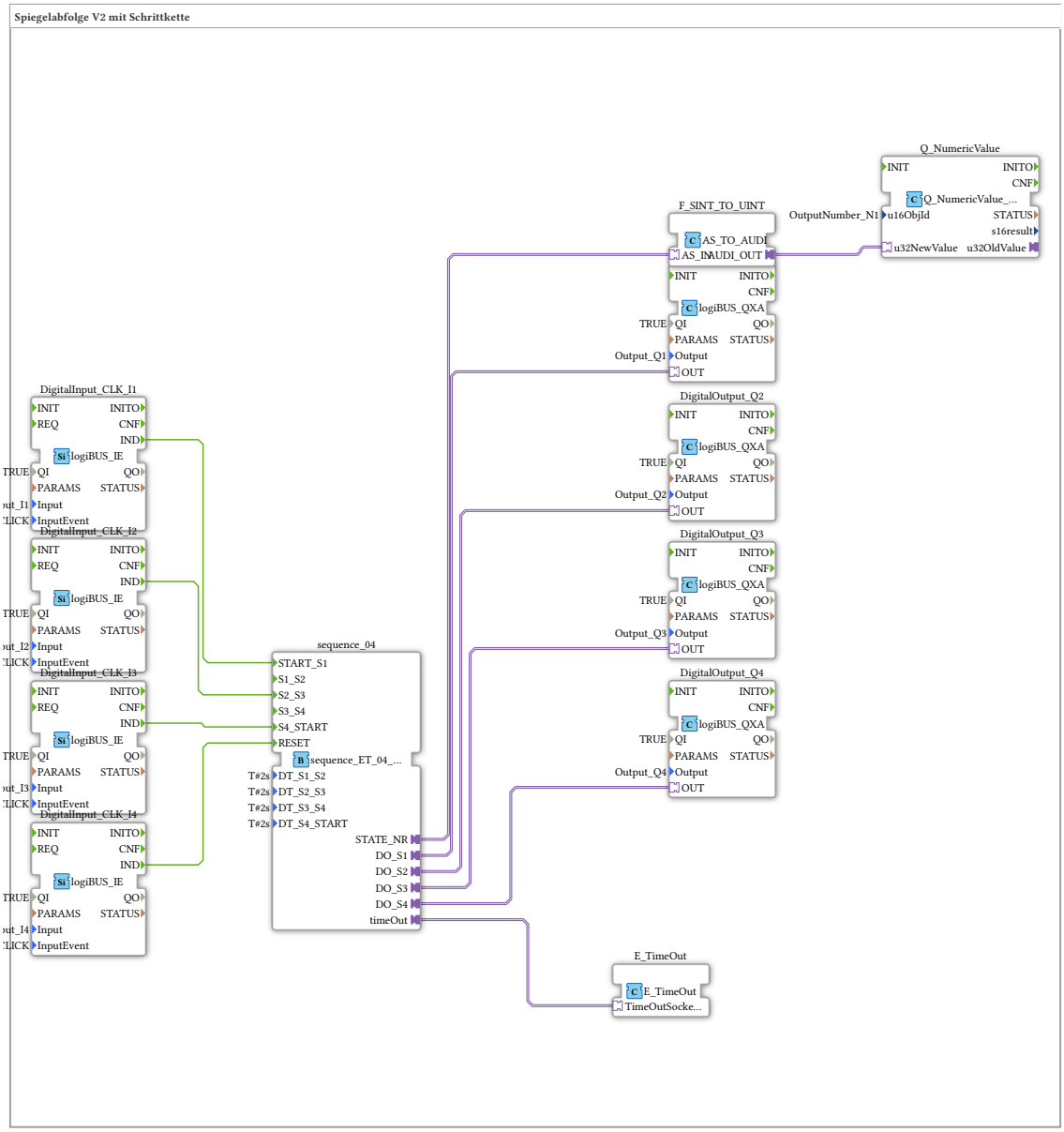
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_034b_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.283 Uebung_035_AX: Spiegelabfolge V2 mit Schrittkette

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_035_AX (Spiegelabfolge V2 mit Schrittkette).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Spiegelabfolge V2 mit Schrittkette**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_035_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
- **E_TimeOut**: Instanz des Typs iec61499::events::E_TimeOut.
- **DigitalOutput_Q2**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q2
- **DigitalOutput_Q3**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q3

- **DigitalOutput_Q4**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q4
- **DigitalInput_CLK_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **DigitalInput_CLK_I2**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I2
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **DigitalInput_CLK_I3**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I3
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **DigitalInput_CLK_I4**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I4
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **Q_NumericValue**: Instanz des Typs isobus::UT::Q::Q_NumericValue_AUDI.
  - Parameter u16ObjId = OutputNumber_N1
- **F_SINT_TO_UINT**: Instanz des Typs adapter::conversion::unidirectional::AS_TO_AUDI.
- **sequence_04**: Instanz des Typs logiBUS::utils::sequence::combi::sequence_ET_04_AX.
  - Parameter DT_S1_S2 = T#2s
  - Parameter DT_S2_S3 = T#2s
  - Parameter DT_S3_S4 = T#2s
  - Parameter DT_S4_START = T#2s

Verbindungen und Schnittstellen

Adapterverbindungen:

- sequence_04.timeOut -> E_TimeOut.TimeOutSocket
- sequence_04.STATE_NR -> F_SINT_TO_UINT.AS_IN
- sequence_04.DO_S1 -> DigitalOutput_Q1.OUT
- sequence_04.DO_S2 -> DigitalOutput_Q2.OUT
- sequence_04.DO_S3 -> DigitalOutput_Q3.OUT
- sequence_04.DO_S4 -> DigitalOutput_Q4.OUT
- F_SINT_TO_UINT.AUDI_OUT -> Q_NumericValue.u32NewValue

Ereignisverbindungen:

- DigitalInput_CLK_I1.IND -> sequence_04.START_S1
- DigitalInput_CLK_I2.IND -> sequence_04.S2_S3
- DigitalInput_CLK_I3.IND -> sequence_04.S4_START
- DigitalInput_CLK_I4.IND -> sequence_04.RESET

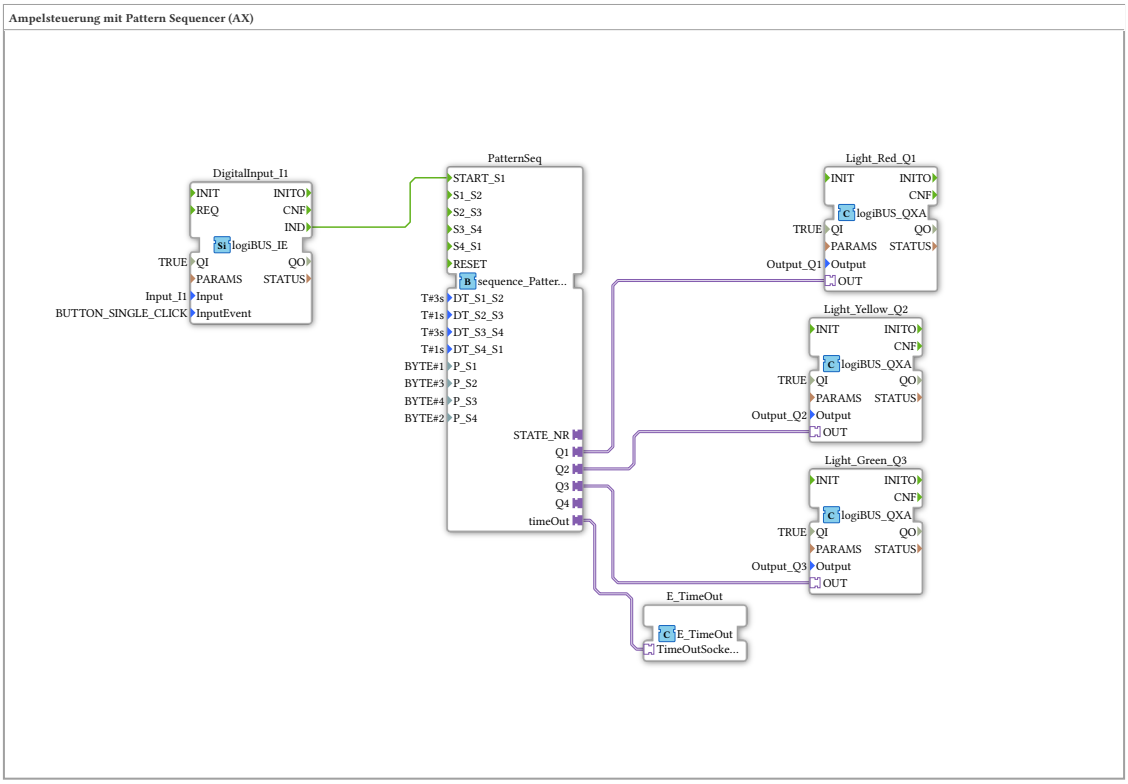
Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_035_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.284 Uebung_035a1_AX: Ampelsteuerung mit Pattern Sequencer (AX)



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung implementiert eine **Ampelsteuerung** unter Verwendung eines **Pattern Sequencers** (Muster-Ablaufsteuerung). Das Ziel ist es, eine klassische Verkehrsampel-Sequenz (Rot -> Rot/Gelb -> Grün -> Gelb -> Rot) durch definierte Zeitintervalle und Bitmuster zu steuern. Die Übung nutzt die Adapter-Technologie (AX) für die Anbindung der Ausgänge und des Timings.

Verwendete Funktionsbausteine (FBs)

In dieser Übung werden verschiedene Bausteine innerhalb des `SubAppNetwork` verschaltet. Nachfolgend sind die wichtigsten Komponenten detailliert beschrieben.

Sub-Bausteine: PatternSeq

Dieser Baustein ist das Herzstück der Steuerung und regelt den zeitlichen Ablauf sowie die Ausgabemuster der Ampelphasen.

- **Typ**: `logiBUS::utils::sequence::pattern::sequence_Pattern_04_04_loop_AX`
- **Verwendete interne Parameter**:
  - **DT_S1_S2** = `T#3s` (Dauer der Phase 1: Rot)
  - **DT_S2_S3** = `T#1s` (Dauer der Phase 2: Rot-Gelb)
  - **DT_S3_S4** = `T#3s` (Dauer der Phase 3: Grün)
  - **DT_S4_S1** = `T#1s` (Dauer der Phase 4: Gelb)
  - **P_S1** = 1 (Muster Phase 1: 001 -> Q1 aktiv)
  - **P_S2** = 3 (Muster Phase 2: 011 -> Q1 & Q2 aktiv)
  - **P_S3** = 4 (Muster Phase 3: 100 -> Q3 aktiv)
  - **P_S4** = 2 (Muster Phase 4: 010 -> Q2 aktiv)
- **Ereignis-/Adapteranschlüsse**:
  - **START_S1**: Startet die Sequenz bei Schritt 1.
  - **Q1, Q2, Q3**: Adapter-Ausgänge zur Steuerung der Lampen.
  - **timeOut**: Adapter für die Verbindung zum Timer-Service.
- **Funktionsweise**: Der Baustein durchläuft zyklisch 4 Schritte. Für jeden Schritt wird ein Bitmuster (`P_SX`) an den Ausgängen angelegt und für die definierte Zeit (`DT_SX_SY`) gehalten.

Sub-Bausteine: DigitalInput_I1

Dieser Baustein verarbeitet das Eingangssignal zum Starten der Ampelsequenz.

- **Typ**: `logiBUS::io::DI::logiBUS_IE`
- **Konfiguration**:
  - **Parameter**: `Input = Input_I1`
  - **Parameter**: `InputEvent = BUTTON_SINGLE_CLICK`
- **Funktionsweise**: Erfasst einen Einzelklick auf dem Eingang I1 und löst ein Event (`IND`) aus.

Sub-Bausteine: Ausgänge (Light_Red, Light_Yellow, Light_Green)

Diese Bausteine stellen die physischen Ausgänge der Ampel dar.

- **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
- **Instanzen**:
  - **Light_Red_Q1**: Verbunden mit `Output_Q1` (Rot)
  - **Light_Yellow_Q2**: Verbunden mit `Output_Q2` (Gelb)
  - **Light_Green_Q3**: Verbunden mit `Output_Q3` (Grün)
- **Funktionsweise**: Sie empfangen Signale über Adapterverbindungen und schalten die entsprechenden Hardware-Ausgänge.

Sub-Bausteine: E_TimeOut

Stellt die Timer-Funktionalität für den Sequenzer bereit.

- **Typ**: `iec61499::events::E_TimeOut`
- **Funktionsweise**: Dient als Service-Interface, um die Zeitverzögerungen der Sequenzschritte zu realisieren.

Programmablauf und Verbindungen

Der Ablauf der Ampelsteuerung gestaltet sich wie folgt:

1. **Startbedingung**: Das Programm wartet auf ein Signal vom Baustein `DigitalInput_I1`. Wenn der Taster an `Input_I1` einfach geklickt wird (`BUTTON_SINGLE_CLICK`), sendet der Ausgang `IND` ein Event an den Eingang `START_S1` des `PatternSeq`-Bausteins.
2. **Sequenzablauf (Pattern Sequencer)**: Der `PatternSeq` Baustein steuert die Ampelphasen basierend auf den konfigurierten Parametern. Die Ausgänge werden binär kodiert angesteuert (Q3, Q2, Q1):
   - **Phase 1 (Rot)**: Dauer 3s (`DT_S1_S2`). Parameter `P_S1 = 1` (Binär `001`) aktiviert Adapter-Ausgang `Q1` -> **Rote Lampe**.
   - **Phase 2 (Rot-Gelb)**: Dauer 1s (`DT_S2_S3`). Parameter `P_S2 = 3` (Binär `011`) aktiviert `Q1` und `Q2` -> **Rote und Gelbe Lampen**.
   - **Phase 3 (Grün)**: Dauer 3s (`DT_S3_S4`). Parameter `P_S3 = 4` (Binär `100`) aktiviert `Q3` -> **Grüne Lampe**.
   - **Phase 4 (Gelb)**: Dauer 1s (`DT_S4_S1`). Parameter `P_S4 = 2` (Binär `010`) aktiviert `Q2` -> **Gelbe Lampe**.
3. **Verbindungen**:
   - Die Logik verwendet **Adapter-Connections** (erkennbar am `logiBUS_QXA` Typ und den verschachtelten Verbindungen), was die Verdrahtung im Diagramm übersichtlicher macht, da Daten und Events gebündelt übertragen werden.
   - Der `E_TimeOut` Baustein ist über den Adapter `timeOut` mit dem Sequenzer verbunden, um die Timer-Events (`T#3s`, `T#1s` etc.) intern zu verarbeiten.

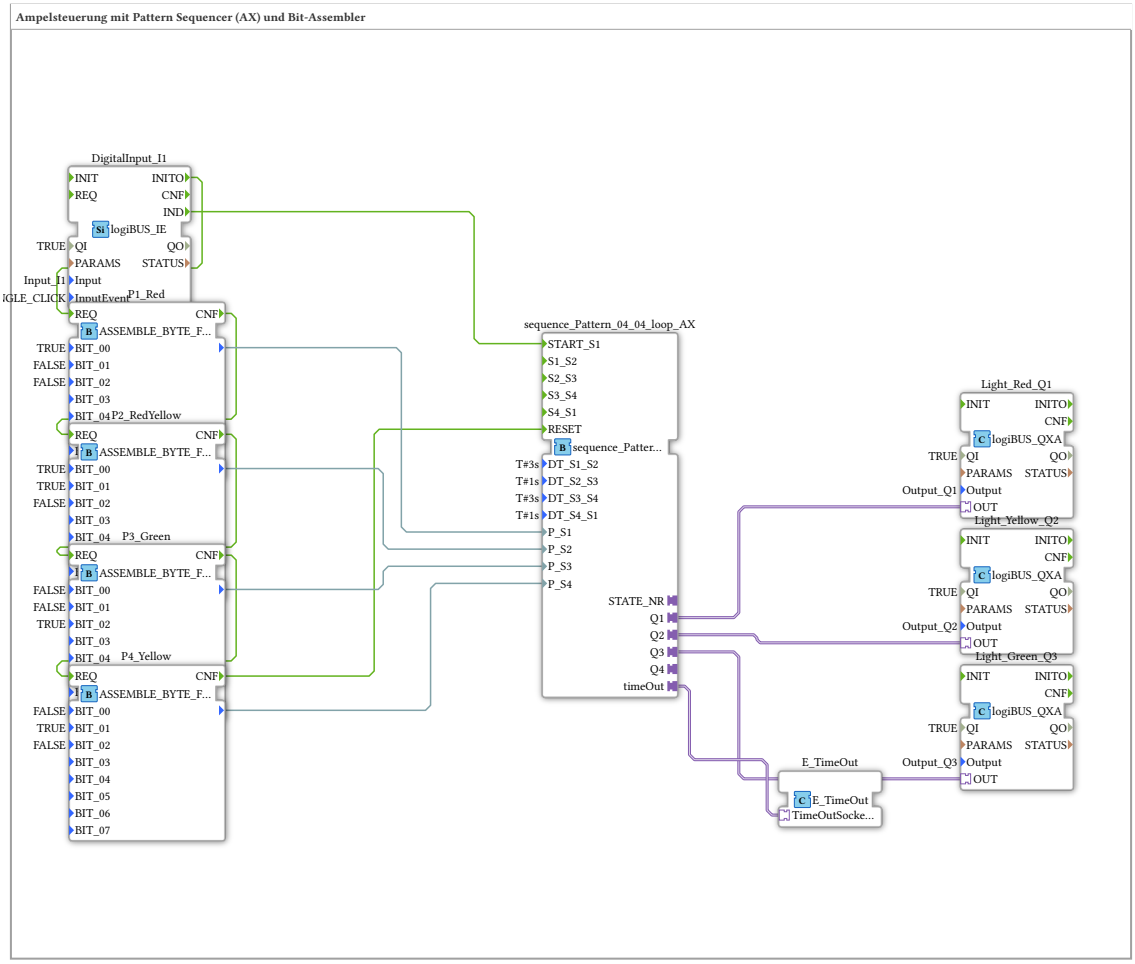
Zusammenfassung

Die Übung `Uebung_035a1_AX` demonstriert effizient, wie komplexe Zustandsautomaten wie eine Ampelsteuerung mit Hilfe eines **Pattern Sequencers** vereinfacht werden können. Anstatt jeden Zustandsübergang einzeln zu programmieren, werden Phasenzeiten und Ausgabemuster parametriert. Die Verwendung von `logiBUS` Adaptern (AX/QXA) zeigt zudem eine moderne Art der Baustein-Kommunikation in 4diac.

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

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

2.1.2.285 Uebung_035a1b_AX: Ampelsteuerung mit Pattern Sequencer (AX) und Bit-Assembler



• – ◦ • – ◦ • – ◦ •

Einleitung

Die Übung **Uebung_035a1b_AX** implementiert eine klassische Ampelsteuerung. Hierbei wird ein **Pattern Sequencer** (Muster-Ablaufsteuerung) im „Loop"-Modus (Schleifenbetrieb) verwendet. Die Besonderheit dieser Übung liegt in der Definition der Ampelphasen: Anstatt die Ausgänge direkt im Sequencer hart zu kodieren, werden die Bitmuster für die einzelnen Phasen (Rot, Rot-Gelb, Grün, Gelb) mithilfe von **Bit-Assemblern** (`ASSEMBLE_BYTE_FROM_BOOLS`) dynamisch erzeugt und dem Sequencer als Parameter übergeben.

Verwendete Funktionsbausteine (FBs)

In dieser SubApplikation werden verschiedene Bausteine verschaltet, um die Steuerungslogik zu realisieren.

Haupt-Steuerungsbaustein: `sequence_Pattern_04_04_loop_AX`

Dieser Baustein ist der Kern der Steuerung. Er durchläuft 4 Schritte in einer Endlosschleife.

- **Typ**: `logiBUS::utils::sequence::pattern::sequence_Pattern_04_04_loop_AX`
- **Funktion**: Er schaltet die Ausgänge basierend auf Byte-Mustern (`P_S1` bis `P_S4`) und hält jeden Schritt für eine definierte Zeit.
- **Konfiguration**:
  - `DT_S1_S2` = `T#3s` (Dauer Rotphase)
  - `DT_S2_S3` = `T#1s` (Dauer Rot-Gelb-Phase)
  - `DT_S3_S4` = `T#3s` (Dauer Grünphase)
  - `DT_S4_S1` = `T#1s` (Dauer Gelbphase)

Muster-Erzeugung: `ASSEMBLE_BYTE_FROM_BOOLS`

Es werden vier Instanzen dieses Bausteins verwendet, um die Lichtmuster für die vier Ampelphasen zu definieren. Jeder Baustein wandelt einzelne Boolesche Signale in ein Byte um, das vom Sequencer interpretiert wird.

1. **Instanz `P1_Red`**:
   - **Zweck**: Definiert die Rot-Phase.
   - **Parameter**: `BIT_00` = `TRUE` (Rot aktiv).
2. **Instanz `P2_RedYellow`**:
   - **Zweck**: Definiert die Rot-Gelb-Phase.
   - **Parameter**: `BIT_00` = `TRUE` (Rot), `BIT_01` = `TRUE` (Gelb).
3. **Instanz `P3_Green`**:
   - **Zweck**: Definiert die Grün-Phase.
   - **Parameter**: `BIT_02` = `TRUE` (Grün).
4. **Instanz `P4_Yellow`**:
   - **Zweck**: Definiert die Gelb-Phase.
   - **Parameter**: `BIT_01` = `TRUE` (Gelb).

Ein-/Ausgabe-Bausteine

- **`DigitalInput_I1`** (`logiBUS::io::DI::logiBUS_IE`):
  - Repräsentiert den Taster zum Starten der Sequenz.
  - Reagiert auf das Ereignis `BUTTON_SINGLE_CLICK`.
- **`Light_Red_Q1`**, **`Light_Yellow_Q2`**, **`Light_Green_Q3`** (`logiBUS::io::DQ::logiBUS_QXA`):
  - Repräsentieren die physischen Ampelleuchten.
  - Werden über Adapter-Verbindungen (`OUT`) vom Sequencer angesteuert.

Programmablauf und Verbindungen

Der Ablauf der Steuerung gliedert sich in die Initialisierung der Muster und den eigentlichen Zyklus der Ampel.

1. Initialisierung

Damit der Sequencer weiß, welche Lampen in welchem Schritt leuchten sollen, müssen die Bitmuster (`P1` bis `P4`) zunächst berechnet und übergeben werden.

- Das Initialisierungs-Signal (`INIT0`) vom Eingangsbaustein `DigitalInput_I1` startet eine Kette.
- Es triggert nacheinander die `REQ`-Eingänge der Assembler-Bausteine: `P1_Red` -> `P2_RedYellow` -> `P3_Green` -> `P4_Yellow`.
- Am Ende dieser Kette wird der `RESET`-Eingang des Sequencers (`sequence_Pattern_04_04_loop_AX`) ausgelöst. Dadurch werden die erzeugten Byte-Muster an die Eingänge `P_S1` bis `P_S4` des Sequencers übernommen.

2. Start und Zyklus

- Ein einfacher Klick (`BUTTON_SINGLE_CLICK`) auf den Eingang `DigitalInput_I1` sendet ein Event an `START_S1` des Sequencers.
- Der Sequencer beginnt nun seinen Ablauf:
  1. **Schritt 1**: Muster von `P1_Red` wird ausgegeben (Lampe Rot an). Wartezeit 3 Sekunden.
  2. **Schritt 2**: Muster von `P2_RedYellow` wird ausgegeben (Rot und Gelb an). Wartezeit 1 Sekunde.
  3. **Schritt 3**: Muster von `P3_Green` wird ausgegeben (Grün an). Wartezeit 3 Sekunden.
  4. **Schritt 4**: Muster von `P4_Yellow` wird ausgegeben (Gelb an). Wartezeit 1 Sekunde.
- Da es sich um einen Loop-Baustein handelt, beginnt der Zyklus nach Schritt 4 wieder automatisch bei Schritt 1.

3. Ausgänge

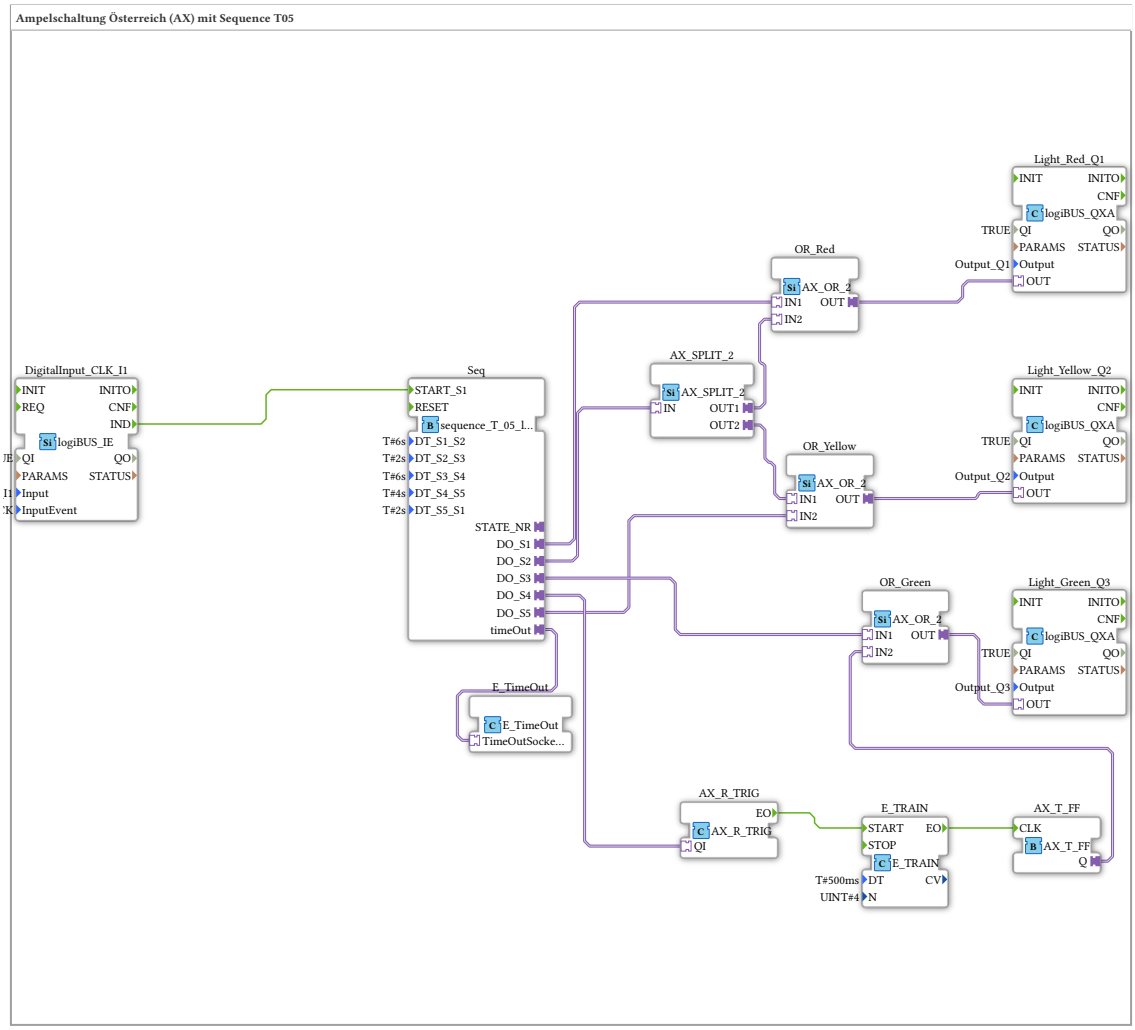
Die Ausgänge des Sequencers (`Q1`, `Q2`, `Q3`) sind als Adapter ausgeführt und direkt mit den Ausgangsbausteinen für die Lampen verbunden. Das Byte-Muster wird intern im Sequencer auf diese Adapter abgebildet (Bit 0 -> Q1, Bit 1 -> Q2, Bit 2 -> Q3).

Zusammenfassung

Diese Übung demonstriert eine fortgeschrittene Methode der Ablaufsteuerung. Durch die Trennung von **Zeitablauf** (Sequencer) und **Zustandsdefinition** (Bit-Assembler) wird der Code modularer. Änderungen

an den Ampelphasen (z.B. welche Lampen leuchten) können einfach in den `ASSEMBLE_BYTE_FROM_BOOLS` Bausteinen vorgenommen werden, ohne die Logik des Sequencers selbst ändern zu müssen. Die Steuerung bildet den typischen deutschen Ampelzyklus ab.

2.1.2.286 Uebung_035a2_AX: Ampelschaltung Österreich (AX) mit Sequence T05



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung implementiert eine **Ampelschaltung nach österreichischem Vorbild (AX)** unter Verwendung des Standards IEC 61499. Im Gegensatz zur deutschen Ampelschaltung (Rot -> Rot/Gelb -> Grün -> Gelb -> Rot) beinhaltet die österreichische Sequenz eine **Grün-Blinken-Phase** vor dem Umschalten auf Gelb.

Die Steuerung erfolgt über eine zeitgesteuerte Sequenz mit 5 Schritten (`sequence_T_05_loop_AX`), wobei die verschiedenen Phasen logisch mit den Ausgängen für Rot, Gelb und Grün verknüpft sind.

Verwendete Funktionsbausteine (FBs)

Haupt-Steuerungsbaustein: `Seq`

Dieser Baustein steuert den zeitlichen Ablauf der Ampelphasen.

- **Typ**: `logiBUS::utils::sequence::timed::sequence_T_05_loop_AX`
- **Funktionsweise**: Er schaltet nacheinander durch 5 Zustände (S1 bis S5). Die Dauer der Zustände wird über Parameter definiert.
- **Konfigurierte Parameter**:
  - `DT_S1_S2` = `T#6s` (Dauer Phase 1: Rot)

- DT_S2_S3 = T#2s (Dauer Phase 2: Rot + Gelb)
- DT_S3_S4 = T#6s (Dauer Phase 3: Grün)
- DT_S4_S5 = T#4s (Dauer Phase 4: Grün Blinken)
- DT_S5_S1 = T#2s (Dauer Phase 5: Gelb)

Ein- und Ausgabebausteine

- **DigitalInput_CLK_I1** (logiBUS::io::DI::logiBUS_IE):
  - Dient als Startsignal für die Sequenz.
  - Parameter: Reagiert auf BUTTON_SINGLE_CLICK an Eingang Input_I1.
- **Light_Red_Q1, Light_Yellow_Q2, Light_Green_Q3** (logiBUS::io::DQ::logiBUS_QXA):
  - Repräsentieren die physischen Ampelleuchten (Rot, Gelb, Grün).
  - Verknüpft mit Output_Q1, Output_Q2, Output_Q3.

Logik- und Hilfsbausteine

- **OR_Red, OR_Yellow, OR_Green** (adapter::booleanOperators::AX_OR_2):
  - ODER-Gatter, die Signale aus verschiedenen Sequenzschritten zusammenführen (z.B. leuchtet Gelb allein in Phase 5, aber auch zusammen mit Rot in Phase 2).
- **AX_SPLIT_2** (adapter::events::unidirectional::AX_SPLIT_2):
  - Teilt ein Signal auf zwei Pfade auf. Wird verwendet, um in Phase 2 gleichzeitig Rot und Gelb anzusteuern.
- **E_TimeOut** (iec61499::events::E_TimeOut):
  - Behandelt das Timing für die Sequenz.
- **Blink-Logik für Grün**:
  - **AX_R_TRIG** (adapter::events::unidirectional::AX_R_TRIG): Erkennt die steigende Flanke (Start der Blink-Phase).
  - **E_TRAIN** (iec61499::events::E_TRAIN): Erzeugt eine Serie von Ereignissen (Impulsen).
    - Parameter DT = T#500ms (Intervall).
    - Parameter N = 4 (Anzahl der Impulse).
  - **AX_T_FF** (adapter::events::unidirectional::AX_T_FF): Toggle-Flip-Flop, das durch die Impulse von E_TRAIN den Ausgang umschaltet (Blinken).

Programmablauf und Verbindungen

Das Programm wird durch einen Klick auf den Taster (Input_I1) gestartet, der das Event START_S1 im Sequenz-FB Seq auslöst. Der Ablauf ist wie folgt:

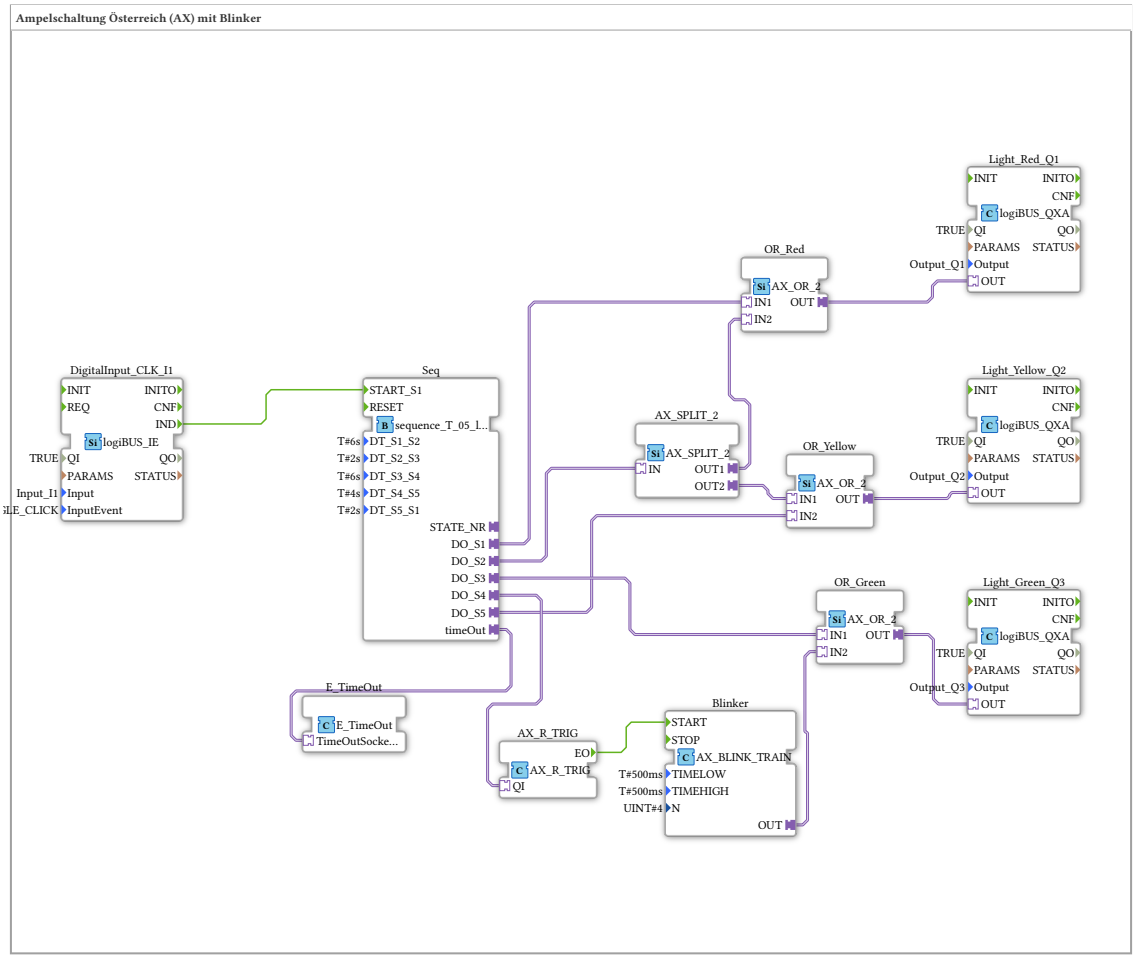
1. **Phase 1 (Rot - 6s):**
   - Seq aktiviert Ausgang DO_S1.
   - Signal geht an OR_Red -> Light_Red_Q1 leuchtet.
2. **Phase 2 (Rot & Gelb - 2s):**
   - Seq aktiviert Ausgang DO_S2.
   - Signal geht an AX_SPLIT_2.
   - AX_SPLIT_2 sendet Signal an OR_Red (Rot bleibt an) und OR_Yellow (Gelb geht an).
3. **Phase 3 (Grün - 6s):**
   - Seq aktiviert Ausgang DO_S3.
   - Signal geht an OR_Green -> Light_Green_Q3 leuchtet konstant.
4. **Phase 4 (Grün Blinken - 4s):**
   - Seq aktiviert Ausgang DO_S4.
   - Das Signal triggert AX_R_TRIG, welcher den E_TRAIN Baustein startet.
   - E_TRAIN sendet Impulse an das Toggle-Flip-Flop AX_T_FF.
   - Der Ausgang Q des Flip-Flops wechselt im 500ms Takt seinen Zustand und ist mit OR_Green verbunden.
   - Ergebnis: Die grüne Lampe blinkt 4-mal (gesteuert durch Parameter N=4).
5. **Phase 5 (Gelb - 2s):**
   - Seq aktiviert Ausgang DO_S5.
   - Signal geht an OR_Yellow -> Light_Yellow_Q2 leuchtet.

Nach Ablauf von Phase 5 ist ein Zyklus beendet. Je nach interner Implementierung des `sequence_T_05_loop_AX` Bausteins startet die Sequenz neu oder wartet auf ein erneutes Eingangssignal.

Zusammenfassung

Die Übung `Uebung_035a2_AX` demonstriert eine komplexe Ampelsteuerung mit spezifischer Ländervariante (Österreich). Besonderes Augenmerk liegt auf der Verwendung von Adapter-Bausteinen (`AX_SPLIT`, `AX_OR`) zur Signalverteilung sowie der Konstruktion eines Blinkgebers mithilfe von `E_TRAIN` und `AX_T_FF`, um das „Grünblinken“ vor der Gelbphase zu realisieren.

2.1.2.287 Uebung_035a3_AX: Ampelschaltung Österreich (AX) mit Blinker



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung implementiert eine Ampelschaltung nach österreichischem Vorbild („Ampelschaltung Österreich (AX) mit Blinker“). Im Gegensatz zu Standard-Ampelschaltungen beinhaltet diese Sequenz die Phase „Grün-Blinken“ vor dem Wechsel auf Gelb sowie die Phase Rot-Gelb vor dem Wechsel auf Grün. Die Steuerung erfolgt über einen sequenziellen Ablaufbaustein, der durch einen Taster gestartet wird.

Verwendete Funktionsbausteine (FBs)

In dieser SubApplikation werden verschiedene logische Bausteine, Zeitglieder und I/O-Treiber verwendet, um die Ampelphasen zu steuern.

Haupt-Steuerbausteine

Seq (`logiBUS::utils::sequence::timed::sequence_T_05_loop_AX`)

Dieser Baustein ist der Kern der Ablaufsteuerung. Er definiert 5 zeitgesteuerte Zustände (S1 bis S5), die in einer Schleife durchlaufen werden.

- **Typ**: Sequenz-Controller (Timed Loop)
- **Parameter**:
  - `DT_S1_S2` = `T#6s` (Dauer Phase 1: Rot)
  - `DT_S2_S3` = `T#2s` (Dauer Phase 2: Rot-Gelb)
  - `DT_S3_S4` = `T#6s` (Dauer Phase 3: Grün)
  - `DT_S4_S5` = `T#4s` (Dauer Phase 4: Grün-Blinken)
  - `DT_S5_S1` = `T#2s` (Dauer Phase 5: Gelb)
- **Funktionsweise**: Nach Aktivierung durch `START_S1` werden die Ausgänge `DO_S1` bis `DO_S5` nacheinander für die definierte Zeitdauer aktiv geschaltet.

Blinker (`adapter::events::unidirectional::signals::AX_BLINK_TRAIN`)

Dieser Baustein erzeugt das Blinksignal für die Grün-Phase.

- **Typ**: Signalgenerator / Blinker
- **Parameter**:
  - `TIMELOW` = `T#500ms` (Aus-Zeit)
  - `TIMEHIGH` = `T#500ms` (Ein-Zeit)
  - `N` = `4` (Anzahl der Blinkimpulse)
- **Funktionsweise**: Sobald der Baustein getriggert wird, sendet er 4 Impulse (je 500ms an/aus) an den Ausgang. Dies realisiert das für Österreich typische 4-malige Blinken am Ende der Grünphase.

Logik- und Hilfsbausteine

- **OR_Red, OR_Yellow, OR_Green** (`adapter::booleanOperators::AX_OR_2`): ODER-Verknüpfungen, um verschiedene Sequenzschritte auf dieselbe Lampe zu leiten (z.B. Rot leuchtet allein in S1, aber auch zusammen mit Gelb in S2).
- **AX_SPLIT_2** (`adapter::events::unidirectional::AX_SPLIT_2`): Ein Signalverteiler, der ein Eingangssignal auf zwei Pfade aufteilt.
- **AX_R_TRIG** (`adapter::events::unidirectional::AX_R_TRIG`): Flankenerkennung (Rising Trigger), um den Blinker präzise zu starten.
- **E_TimeOut** (`iec61499::events::E_TimeOut`): Behandelt Timeouts der Sequenz.

Ein-/Ausgabe Bausteine

- **DigitalInput_CLK_I1** (`logiBUS::io::DI::logiBUS_IE`): Liest den Taster `Input_I1` (Event: `BUTTON_SINGLE_CLICK`).
- **Light_Red_Q1** (`logiBUS::io::DQ::logiBUS_QXA`): Steuert die rote Lampe (`Output_Q1`).
- **Light_Yellow_Q2** (`logiBUS::io::DQ::logiBUS_QXA`): Steuert die gelbe Lampe (`Output_Q2`).
- **Light_Green_Q3** (`logiBUS::io::DQ::logiBUS_QXA`): Steuert die grüne Lampe (`Output_Q3`).

Programmablauf und Verbindungen

Der Ablauf wird durch einen Einzelklick auf den Taster (`Input_I1`) gestartet, welcher das Event `START_S1` am Sequenzbaustein **Seq** auslöst. Daraufhin läuft folgende Schleife ab:

1. **Phase Rot (S1 - 6s)**:
   - Ausgang `DO_S1` ist aktiv.
   - Signal geht an `OR_Red` -> `Light_Red_Q1` (Rot an).
2. **Phase Rot-Gelb (S2 - 2s)**:
   - Ausgang `DO_S2` ist aktiv.
   - Signal geht an `AX_SPLIT_2`.
   - Von dort wird es aufgeteilt auf:
     - `OR_Red` -> `Light_Red_Q1` (Rot bleibt an).
     - `OR_Yellow` -> `Light_Yellow_Q2` (Gelb geht an).
3. **Phase Grün (S3 - 6s)**:
   - Ausgang `DO_S3` ist aktiv.
   - Signal geht an `OR_Green` -> `Light_Green_Q3` (Grün an).
4. **Phase Grün-Blinken (S4 - 4s)**:
   - Ausgang `DO_S4` ist aktiv.
   - Signal triggert über `AX_R_TRIG` den Baustein **Blinker**.

- Der Blinker sendet eine Impulsfolge an `OR_Green`.
- Ergebnis: Die grüne Lampe (`Light_Green_Q3`) blinkt 4-mal (insgesamt 4 Sekunden).

5. **Phase Gelb (S5 - 2s)**:
   - Ausgang `DO_S5` ist aktiv.
   - Signal geht an `OR_Yellow` -> `Light_Yellow_Q2` (Gelb an).

Nach Phase 5 beginnt der Zyklus wieder bei Phase 1 (Rot).

Zusammenfassung

Diese Übung demonstriert die Erstellung einer komplexeren Verkehrsampelsteuerung unter Verwendung von logiBUS-Adaptern. Besonderes Augenmerk liegt auf der korrekten Abbildung der österreichischen Signalfolge (inklusive Grün-Blinken und Rot-Gelb-Phase). Lernziele sind der Umgang mit zeitgesteuerten Sequenzbausteinen (`sequence_T_05_loop_AX`), die Verwendung von Signal-Splittern und logischen ODER-Verknüpfungen zur Ansteuerung von Ausgängen aus mehreren Zuständen heraus sowie die Integration eines Blink-Generators.

2.1.2.288 Uebung_035a_AX: Reine Zeitsteuerung 4-Kanal (Adapter Version)

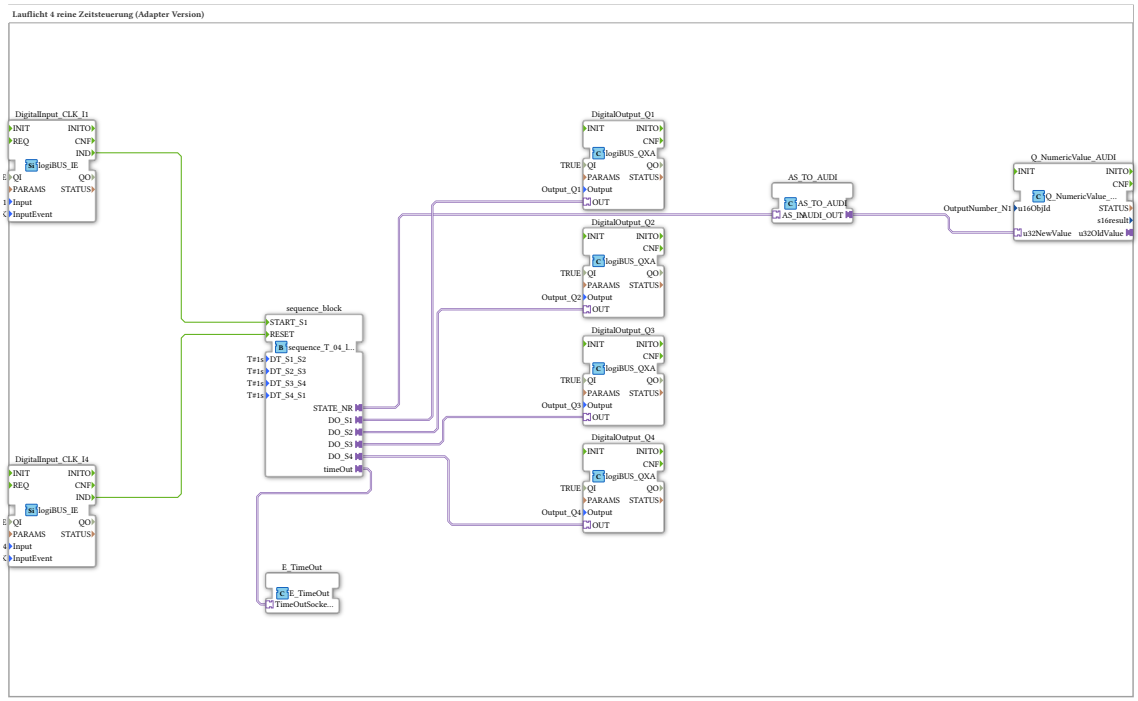
Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_035a_AX`. Wir bauen eine rein zeitgesteuerte Schrittkette mit AX-Adaptertechnologie.

Ziel der Übung

Realisierung einer automatischen Abfolge von 4 Ausgängen (Q1, Q2, Q3, Q4), bei der jeder Schritt ausschließlich nach Ablauf einer festen Wartezeit (`DT_S1_S2` usw., je `T#1s`) zum nächsten wechselt - ganz ohne weitere Tasterbetätigung während des Ablaufs.

Beschreibung und Komponenten

Die Subapplikation `Uebung_035a_AX` nutzt den Sequenzer `sequence_T_04_loop_AX` (AX-Adaptervariante), der 4 Zustände (`S1` bis `S4`) rein zeitgesteuert durchläuft.



- **`sequence_T_04_loop_AX`**: Verwaltet die 4 Schritte, jeder Übergang erfolgt nach Ablauf der jeweiligen `DT_Sx_Sy`-Parameter.
- **`E_TimeOut`**: Überwacht die Sequenz als Watchdog.
- **`Q_NumericValue_AUDI`**: Zeigt die aktuelle Schrittnummer auf dem ISOBUS-Terminal an.

Funktionsweise

1. Start durch Taster `I1` -> Sequenz springt zu `S1`.
2. Jeder Ausgang (`Q1, Q2, Q3, Q4`) wird für die eingestellte Zeit aktiv geschaltet, bevor automatisch zum nächsten Schritt gewechselt wird.
3. Nach dem letzten Schritt kehrt die Sequenz automatisch zu S1 zurück (ENDLOS).
4. Reset durch Taster `I4` -> Alles aus, Sequenz zurück in den Grundzustand.

2.1.2.289 Übung 035a: Lauflicht 4 reine Zeitsteuerung

Thema: Ablaufsteuerungen in der Landtechnik

Situationsbeschreibung

Für ein System in der landwirtschaftlichen Automatisierung (z.B. eine Fassreinigung, eine Klappvorrichtung oder ein Gebläse) soll eine Ablaufsteuerung implementiert werden. Es werden 4 Stufen oder Aktoren nacheinander geschaltet.

Die Steuerung soll über den Baustein `sequence_T_04_loop_AX` realisiert werden.

Funktionsbeschreibung der Ablaufsteuerung

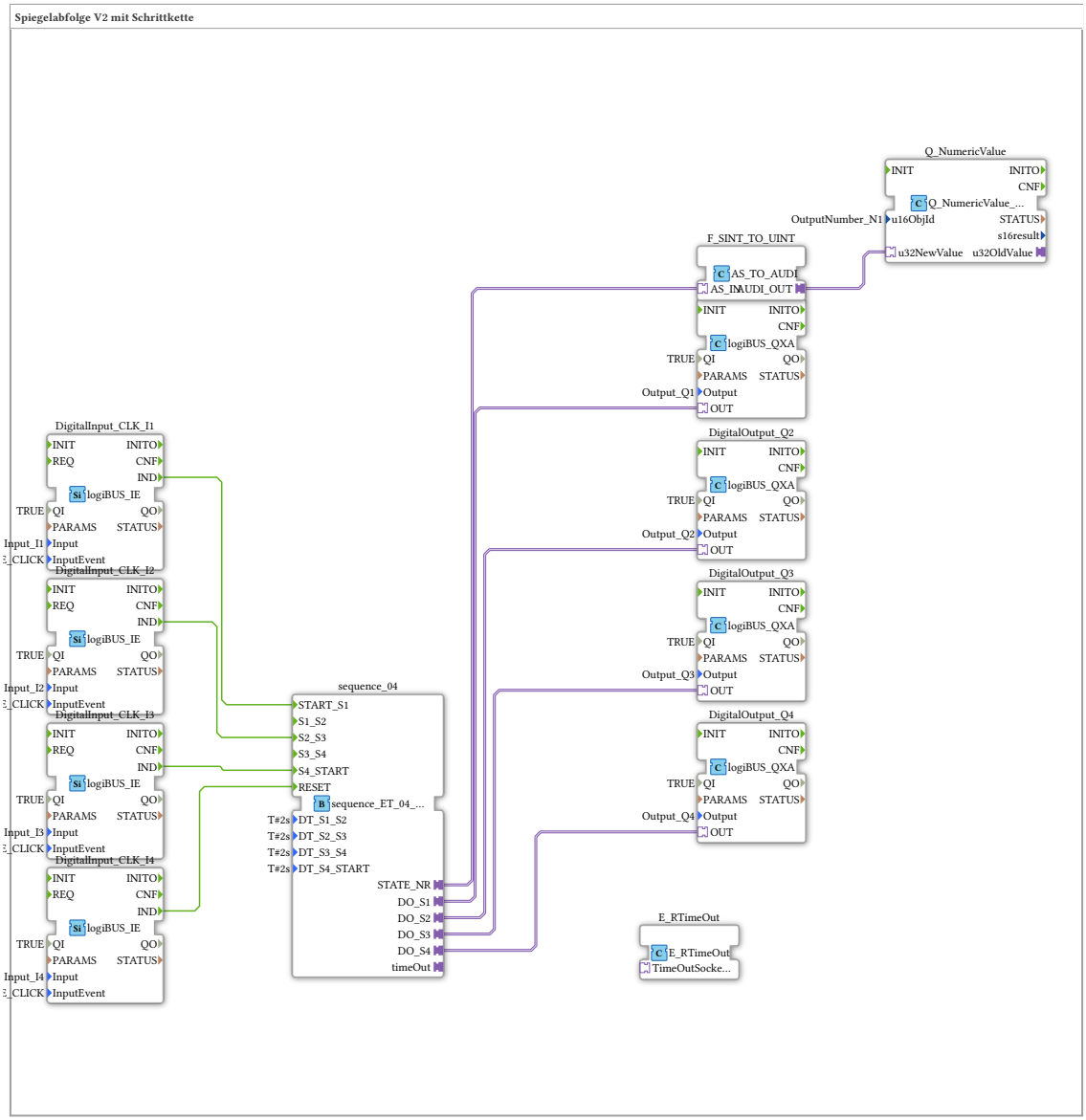
- **Start:** Durch Betätigen des Tasters `I1` (`START_S1`).
- **Phasen:** 4 aufeinanderfolgende Ausgangsschritte (Ausgänge `Q1` bis `Q4`).
- **Übergänge:** Die Schrittübergänge laufen rein zeitgesteuert ab (jeder Schritt dauert 1 Sekunde).
- **Reset:** Zu jedem Zeitpunkt kann die Schrittkette über einen Reset (z.B. Taster `I4` oder Softkey) in den Ausgangszustand (Schritt 0) zurückgesetzt werden.

Arbeitsauftrag

1. Öffnen Sie das 4diac-Projekt und legen Sie die SubApp `Uebung_035a_AX` bzw. `Uebung_035a` an.
2. Platzieren Sie den Sequenzer-Baustein `sequence_T_04_loop_AX` und konfigurieren Sie die Parameter.
3. Verbinden Sie den Start-Taster `I1` und den Reset-Taster `I4` mit den entsprechenden Eingängen.
4. Schalten Sie die 4 Ausgänge auf die physischen Ausgänge `Output_Q1` bis `Output_Q4`.
5. Zur Visualisierung soll die Zustandsnummer `STATE_NR` über den Baustein `AS_TO_AUDI` (Konvertierung `AS` ⟶ `AUDI`) auf dem ISOBUS Virtual Terminal-Ausgang `OutputNumber_N1` angezeigt werden.

2.1.2.290 Uebung_035b_AX: Spiegelabfolge V2 mit Schrittkette

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_035b_AX (Spiegelabfolge V2 mit Schrittkette).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Spiegelabfolge V2 mit Schrittkette**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_035b_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
- **E_RTimeOut**: Instanz des Typs iec61499::events::E_RTimeOut.
- **DigitalOutput_Q2**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q2
- **DigitalOutput_Q3**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q3
- **DigitalOutput_Q4**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.

  - Parameter QI = TRUE
  - Parameter Output = Output_Q4
- **DigitalInput_CLK_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **DigitalInput_CLK_I2**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I2
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **DigitalInput_CLK_I3**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I3
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **DigitalInput_CLK_I4**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I4
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **Q_NumericValue**: Instanz des Typs isobus::UT::Q::Q_NumericValue_AUDI.
  - Parameter u16ObjId = OutputNumber_N1
- **F_SINT_TO_UINT**: Instanz des Typs adapter::conversion::unidirectional::AS_TO_AUDI.
- **sequence_04**: Instanz des Typs logiBUS::utils::sequence::combi::sequence_ET_04_AX.
  - Parameter DT_S1_S2 = T#2s
  - Parameter DT_S2_S3 = T#2s
  - Parameter DT_S3_S4 = T#2s
  - Parameter DT_S4_START = T#2s

Verbindungen und Schnittstellen

Adapterverbindungen:

- sequence_04.STATE_NR -> F_SINT_TO_UINT.AS_IN
- sequence_04.DO_S1 -> DigitalOutput_Q1.OUT
- sequence_04.DO_S2 -> DigitalOutput_Q2.OUT
- sequence_04.DO_S3 -> DigitalOutput_Q3.OUT
- sequence_04.DO_S4 -> DigitalOutput_Q4.OUT
- F_SINT_TO_UINT.AUDI_OUT -> Q_NumericValue.u32NewValue

Ereignisverbindungen:

- DigitalInput_CLK_I1.IND -> sequence_04.START_S1
- DigitalInput_CLK_I2.IND -> sequence_04.S2_S3
- DigitalInput_CLK_I3.IND -> sequence_04.S4_START
- DigitalInput_CLK_I4.IND -> sequence_04.RESET

Hinweise aus dem Modell

TODO! das soll da Beispiel mit E_RT werden, das funktioniert aber noch nicht.

Programmablauf und Funktionsweise

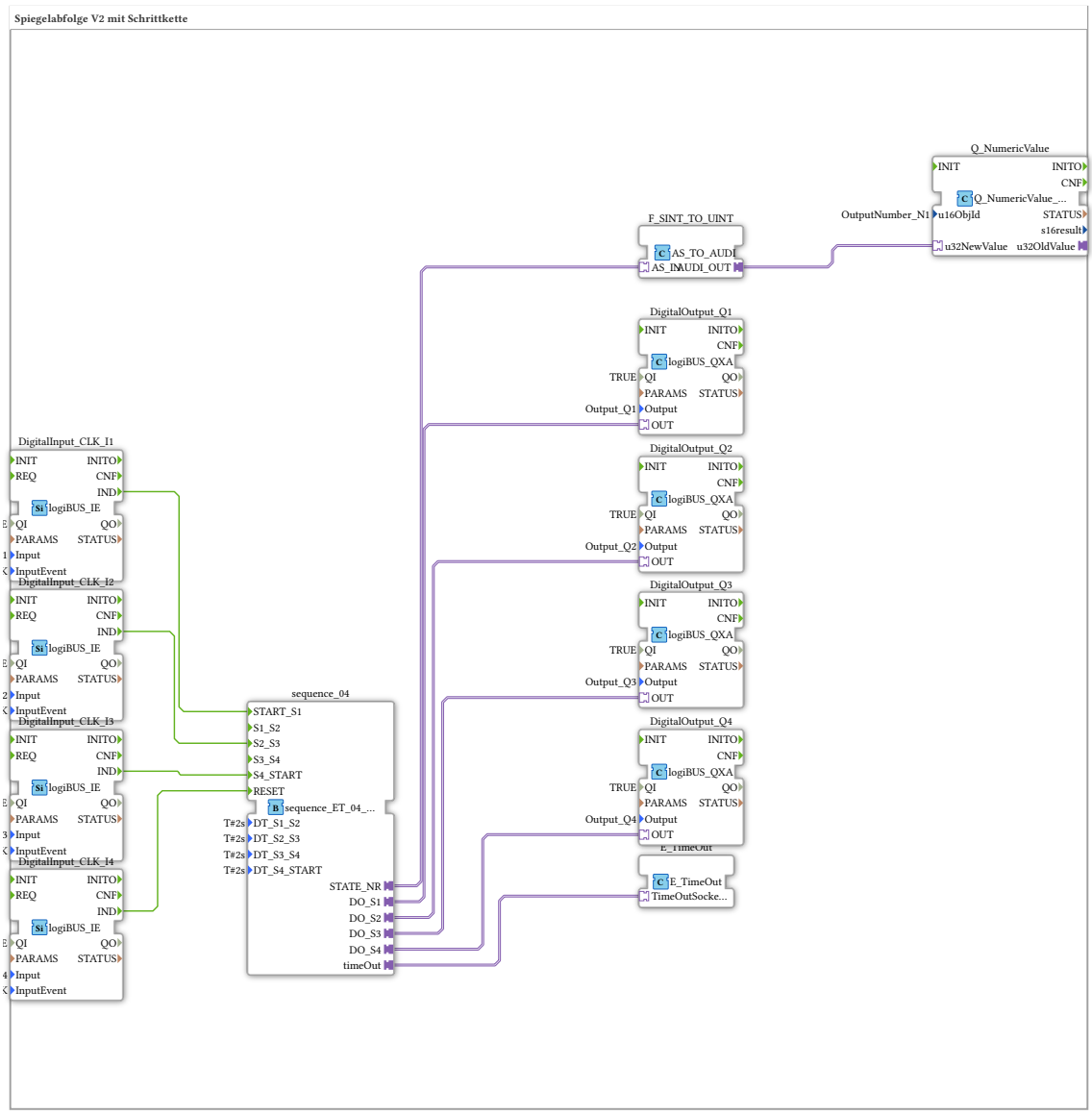
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_035b_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.291 Uebung_035c_AX: Spiegelabfolge V2 mit Schrittkette

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_035c_AX (Spiegelabfolge V2 mit Schrittkette).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Spiegelabfolge V2 mit Schrittkette**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_035c_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
- **E_TimeOut**: Instanz des Typs iec61499::events::E_TimeOut.
- **DigitalOutput_Q2**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q2
- **DigitalOutput_Q3**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.

  - Parameter QI = TRUE
  - Parameter Output = Output_Q3
- **DigitalOutput_Q4**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q4
- **DigitalInput_CLK_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **DigitalInput_CLK_I2**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I2
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **DigitalInput_CLK_I3**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I3
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **DigitalInput_CLK_I4**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I4
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **Q_NumericValue**: Instanz des Typs isobus::UT::Q::Q_NumericValue_AUDI.
  - Parameter u16ObjId = OutputNumber_N1
- **F_SINT_TO_UINT**: Instanz des Typs adapter::conversion::unidirectional::AS_TO_AUDI.
- **sequence_04**: Instanz des Typs logiBUS::utils::sequence::combi::sequence_ET_04_AX.
  - Parameter DT_S1_S2 = T#2s
  - Parameter DT_S2_S3 = T#2s
  - Parameter DT_S3_S4 = T#2s
  - Parameter DT_S4_START = T#2s

Verbindungen und Schnittstellen

Adapterverbindungen:

- sequence_04.timeOut -> E_TimeOut.TimeOutSocket
- sequence_04.STATE_NR -> F_SINT_TO_UINT.AS_IN
- sequence_04.DO_S1 -> DigitalOutput_Q1.OUT
- sequence_04.DO_S2 -> DigitalOutput_Q2.OUT
- sequence_04.DO_S3 -> DigitalOutput_Q3.OUT
- sequence_04.DO_S4 -> DigitalOutput_Q4.OUT
- F_SINT_TO_UINT.AUDI_OUT -> Q_NumericValue.u32NewValue

Ereignisverbindungen:

- DigitalInput_CLK_I1.IND -> sequence_04.START_S1
- DigitalInput_CLK_I2.IND -> sequence_04.S2_S3
- DigitalInput_CLK_I3.IND -> sequence_04.S4_START
- DigitalInput_CLK_I4.IND -> sequence_04.RESET

Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_035c_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.292 Uebung_036_AX: Spiegelabfolge V2 mit Schrittkette

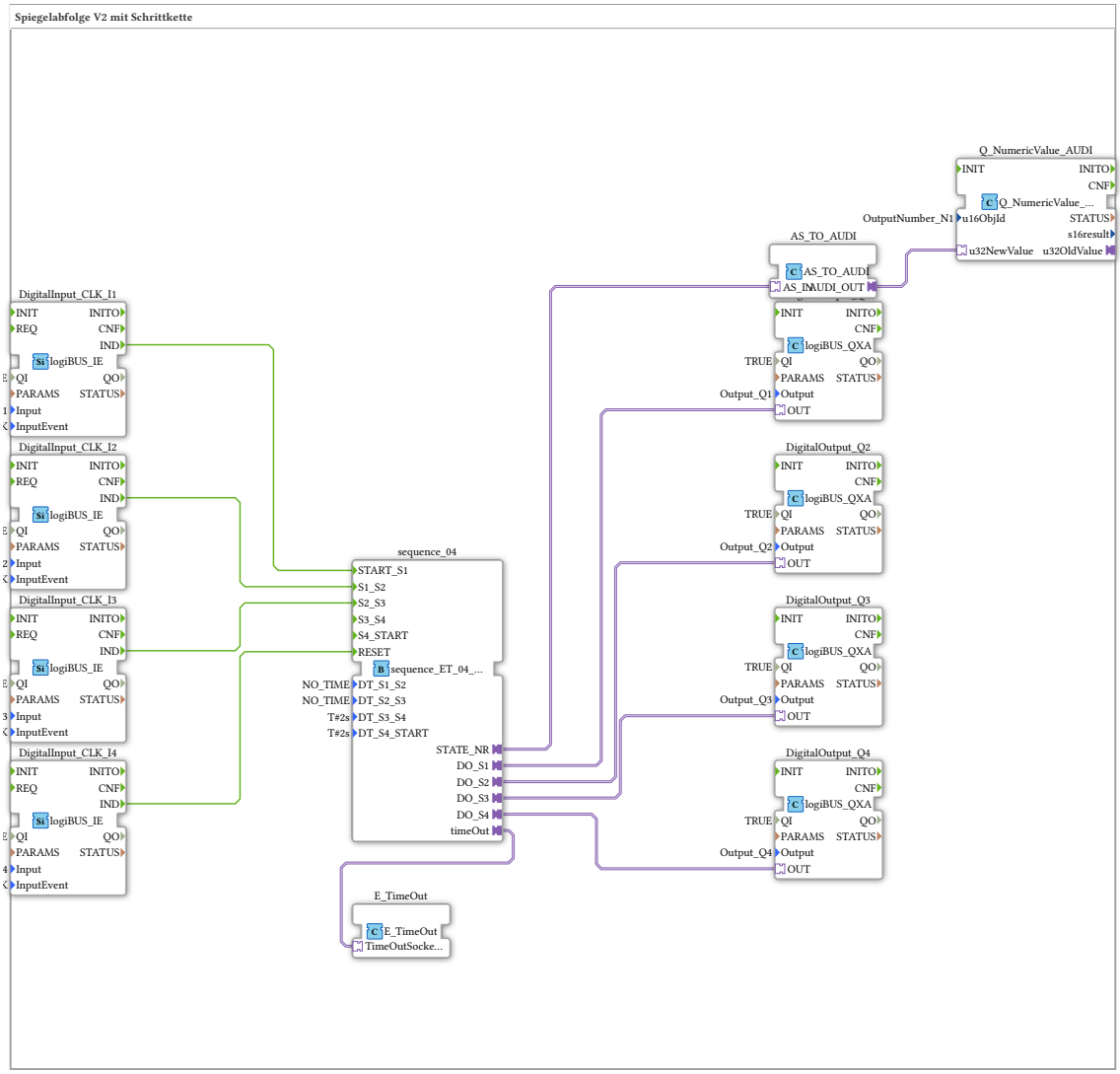
Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_036_AX`. Wir bauen eine 4-Kanal-Schrittkette mit AX-Adaptertechnologie, bei der jeder Schritt manuell per Taster ausgelöst wird.

Ziel der Übung

Realisierung einer Abfolge von 4 Ausgängen (`Q1-Q4`), bei der jeder Übergang durch einen eigenen Taster (`I1-I4`) ausgelöst wird - im Gegensatz zu den zeitgesteuerten Varianten (z. B. `Uebung_037_AX`) läuft hier nichts automatisch weiter.

Beschreibung und Komponenten

Die Subapplikation `Uebung_036_AX` nutzt den kombinierten Sequenzer `sequence_ET_04_AX`, der sowohl Ereignis- als auch Zeiteingänge besitzt. In dieser Übung sind alle vier Übergänge (`START_S1`, `S1_S2`, `S2_S3`, `S3_S4`, `RESET`) direkt mit je einem Taster verdrahtet - die Zeitparameter (`DT_S1_S2`/`DT_S2_S3` = `NO_TIME`, `DT_S3_S4`/`DT_S4_START` = `T#2s`) wirken hier nur als Fallback für die letzten beiden Übergänge, die keinen eigenen Taster mehr haben.



- **`sequence_ET_04_AX`**: 4-Schritt-Sequenzer, hier überwiegend ereignisgesteuert genutzt.
- **`Q_NumericValue_AUDI`**: Zeigt die aktuelle Schrittnummer auf dem ISOBUS-Terminal an.

Funktionsweise

1. Start durch Taster `I1` -> Sequenz springt zu `S1`, `Q1` wird aktiv.
2. Taster `I2` -> Übergang zu `S2`.

3. Taster I3 -> Übergang zu S3.
4. Übergang zu S4 und zurück zum Start erfolgt automatisch nach T#2s, da hierfür kein eigener Taster verdrahtet ist.
5. Reset durch Taster I4 -> Alles aus.

2.1.2.293 Übung 036: Kombinierte 4-Kanal-Schrittkette (Event & Zeit) mit sequence_ET_04_AX

Thema: Ablaufsteuerungen in der Landtechnik

Situationsbeschreibung

Ein Landwirt möchte die Reinigungsprozedur seiner Feldspritze teil-automatisieren, um Wasser zu sparen und den Spülvorgang zu beschleunigen. Das Spülsystem besitzt vier Ventile für verschiedene Spülstufen (Vorspülen, Hauptspülen, Klarspülen, Ausblasen).

Die ersten beiden Phasen (Vorspülen, Hauptspülen) sollen manuell durch den Bediener per Knopfdruck weitergeschaltet werden, da diese je nach Verschmutzung unterschiedlich lange dauern. Die letzten beiden Phasen (Klarspülen, Ausblasen) sollen vollautomatisch für jeweils 2 Sekunden laufen.

Funktionsbeschreibung der Ablaufsteuerung

Die Logik soll mit dem Funktionsbaustein sequence_ET_04_AX.fbt realisiert werden.

Phasen des Spülprozesses

1. **Zustand 0 (Bereit / Idle):** Das System wartet auf den Start. Kein Ventil ist aktiv (STATE_NR = 0).
2. **Schritt 1 (Vorspülen):** Ventil 1 öffnet (DO_S1 wird aktiv). Die Phase läuft unbegrenzt (DT_S1_S2 = NO_TIME). Erst ein Impuls auf Taster I2 (S1_S2) schaltet in Schritt 2.
3. **Schritt 2 (Hauptspülen):** Ventil 2 öffnet (DO_S2 wird aktiv). Die Phase läuft unbegrenzt (DT_S2_S3 = NO_TIME). Erst ein Impuls auf Taster I3 (S2_S3) schaltet in Schritt 3.
4. **Schritt 3 (Klarspülen):** Ventil 3 öffnet (DO_S3 wird aktiv). Die Phase dauert genau **2 Sekunden** (DT_S3_S4 = T#2s) und schaltet danach automatisch weiter.
5. **Schritt 4 (Ausblasen/Abtropfen):** Ventil 4 öffnet (DO_S4 wird aktiv). Die Phase dauert genau **2 Sekunden** (DT_S4_START = T#2s) und schaltet danach automatisch wieder in den Zustand 0 (Bereit).
6. **Reset:** Zu jedem Zeitpunkt kann das System durch Betätigen von Taster I4 (RESET) sofort in den Zustand 0 (Bereit) zurückgesetzt werden.

Arbeitsauftrag

1. Öffnen Sie das 4diac-Projekt und legen Sie die SubApp Uebung_036_AX an (eine Vorlage befindet sich bereits im Ordner Uebungen).
2. Platzieren Sie den Sequenzer-Baustein sequence_ET_04_AX und konfigurieren Sie die Zeitparameter (DT_S1_S2 bis DT_S4_START) entsprechend den Vorgaben.
3. Verbinden Sie den Start-Taster I1 (über das Event BUTTON_SINGLE_CLICK von DigitalInput_CLK_I1) mit dem Eingang START_S1 des Sequenzers.
4. Verbinden Sie die Taster I2 und I3 mit den Weiterschalt-Ereignissen (S1_S2, S2_S3).
5. Verbinden Sie den Reset-Taster I4 mit dem Eingang RESET des Sequenzers.
6. Schalten Sie die vier Ausgänge DO_S1 bis DO_S4 auf die digitalen Ausgänge Output_Q1 bis Output_Q4 der Ventilsteuerung.
7. Zur Visualisierung des aktuellen Schrittes soll die Zustandsnummer STATE_NR über den Baustein AS_TO_AUDI in einen UDINT-Wert (AUDI) konvertiert und an den ISOBUS Virtual Terminal-Ausgang OutputNumber_N1 gesendet werden.
8. Testen Sie das Verhalten der Ablaufsteuerung in der Simulation (FORTE) oder auf der realen Hardware.

2.1.2.294 Uebung_036b_AX: Ablaufsteuerung 4-Kanal (Event & Zeit) (Adapter Version)

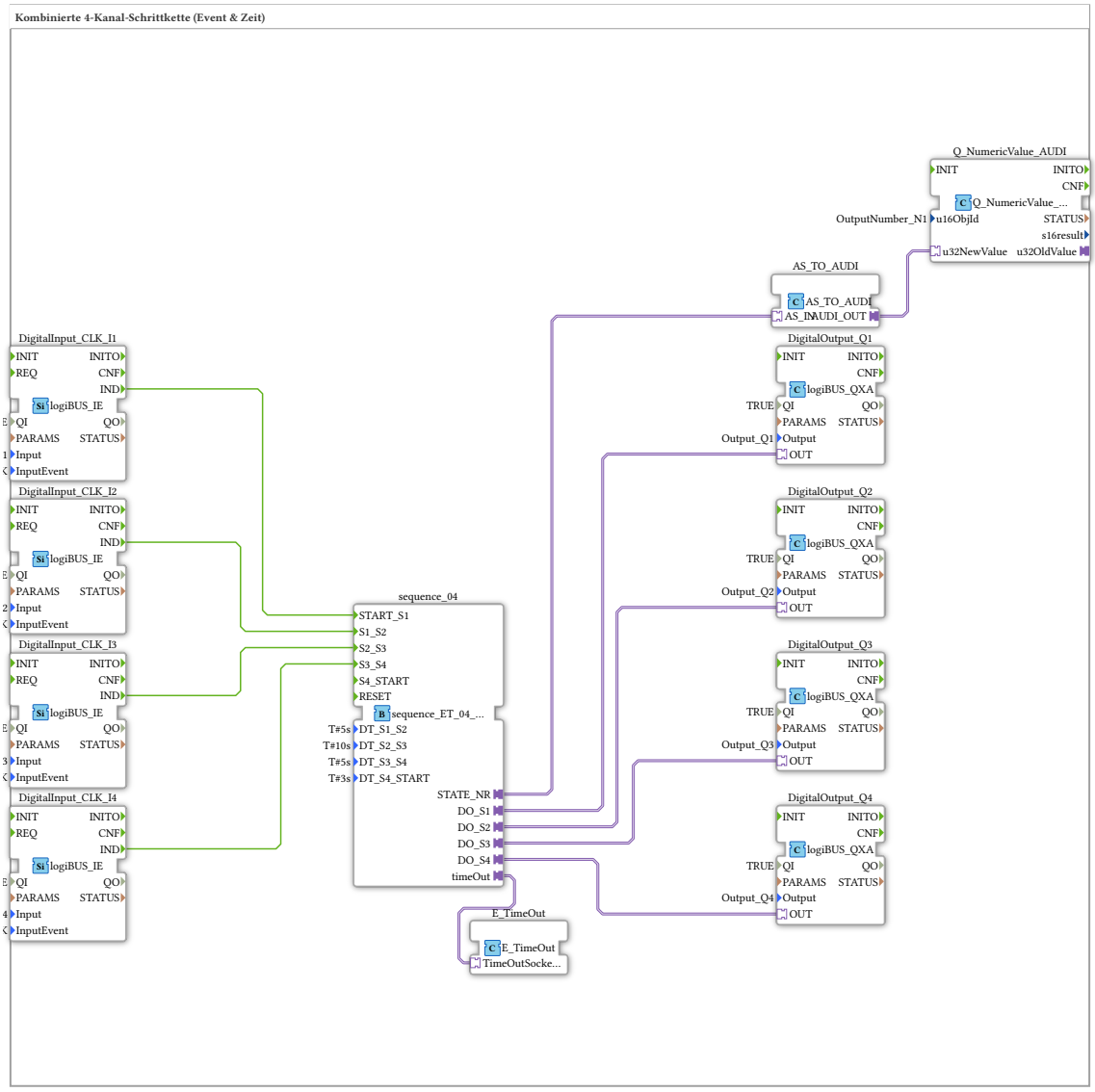
Dieser Artikel beschreibt die logiBUS®-Übung Uebung_036b_AX. Wir bauen eine kombinierte Event- und Zeit-Schrittkette mit AX-Adaptertechnologie.

Ziel der Übung

Realisierung einer Abfolge von 4 Ausgängen (Q1, Q2, Q3, Q4), gestartet und zurückgesetzt über Ereignisse (Taster), wobei die einzelnen Schritte zusätzlich über feste Zeiten (DT_Sx_Sy, je T#1s) weitergeschaltet werden - eine Kombination aus Start/Reset per Taster und automatischem Zeittakt zwischen den Schritten.

Beschreibung und Komponenten

Die Subapplikation `Uebung_036b_AX` nutzt den kombinierten Sequenzer `sequence_ET_04_AX`.



- **`sequence_ET_04_AX`**: Startet auf Ereignis (`I1`), schaltet die 4 Schritte danach zeitgesteuert weiter.
- **`Q_NumericValue_AUDI`**: Zeigt die aktuelle Schrittnummer auf dem ISOBUS-Terminal an.

Funktionsweise

1. Start durch Taster `I1` -> Sequenz springt zu `S1`.
2. Jeder Ausgang (`Q1, Q2, Q3, Q4`) wird für die eingestellte Zeit aktiv geschaltet, bevor automatisch zum nächsten Schritt gewechselt wird.
3. Nach dem letzten Schritt endet die Sequenz.
4. Reset durch Taster `I4` -> Alles aus.

2.1.2.295 Übung 036b: Kombinierte 4-Kanal-Schrittkette (Event & Zeit) mit sequence_ET_04_AX

Thema: Ablaufsteuerungen in der Landtechnik

Situationsbeschreibung

Ein Landwirt möchte die Reinigungsprozedur seiner Feldspritze teil-automatisieren, um Wasser zu sparen und den Spülvorgang zu beschleunigen. Das Spülsystem besitzt vier Ventile für verschiedene Spülstufen

(Vorspülen, Hauptspülen, Klarspülen, Ausblasen). Die Reinigung soll in vier aufeinanderfolgenden Phasen ablaufen. Jede Phase soll eine maximale Dauer haben (Zeitsteuerung), kann jedoch vom Bediener in der Traktorkabine bei Bedarf vorzeitig per Tastendruck auf die nächste Stufe geschaltet werden (Eventsteuerung).

Funktionsbeschreibung der Ablaufsteuerung

Die Logik soll mit dem Funktionsbaustein sequence_ET_04_AX.fbt realisiert werden.

Phasen des Spülprozesses

1. **Zustand 0 (Bereit / Idle):** Das System wartet auf den Start. Kein Ventil ist aktiv (`STATE_NR = 0`).
2. **Schritt 1 (Vorspülen):** Ventil 1 öffnet (`DO_S1` wird aktiv). Maximale Dauer: **5 Sekunden** (`DT_S1_S2 = T#5s`). Ein Impuls auf Taster `I2` (`S1_S2`) schaltet vorzeitig in Schritt 2.
3. **Schritt 2 (Hauptspülen):** Ventil 2 öffnet (`DO_S2` wird aktiv). Maximale Dauer: **10 Sekunden** (`DT_S2_S3 = T#10s`). Ein Impuls auf Taster `I3` (`S2_S3`) schaltet vorzeitig in Schritt 3.
4. **Schritt 3 (Klarspülen):** Ventil 3 öffnet (`DO_S3` wird aktiv). Maximale Dauer: **5 Sekunden** (`DT_S3_S4 = T#5s`). Ein Impuls auf Taster `I4` (`S3_S4`) schaltet vorzeitig in Schritt 4.
5. **Schritt 4 (Ausblasen/Abtropfen):** Ventil 4 öffnet (`DO_S4` wird aktiv). Maximale Dauer: **3 Sekunden** (`DT_S4_START = T#3s`). Nach Ablauf dieser Zeit wechselt das System automatisch wieder in den Zustand 0 (Bereit).

Arbeitsauftrag

1. Öffnen Sie das 4diac-Projekt und legen Sie die SubApp `Uebung_036b_AX` an (eine Vorlage befindet sich bereits im Ordner `Uebungen`).
2. Platzieren Sie den Sequenzer-Baustein `sequence_ET_04_AX` und konfigurieren Sie die Zeitparameter (`DT_S1_S2` bis `DT_S4_START`) entsprechend den Vorgaben.
3. Verbinden Sie den Start-Taster `I1` (über das Event `BUTTON_SINGLE_CLICK` von `DigitalInput_CLK_I1`) mit dem Eingang `START_S1` des Sequenzers.
4. Verbinden Sie die Taster `I2`, `I3` und `I4` mit den entsprechenden Weiterschalt-Ereignissen (`S1_S2`, `S2_S3`, `S3_S4`).
5. Schalten Sie die vier Ausgänge `DO_S1` bis `DO_S4` auf die digitalen Ausgänge `Output_Q1` bis `Output_Q4` der Ventilsteuerung.
6. Zur Visualisierung des aktuellen Schrittes soll die Zustandsnummer `STATE_NR` über den Baustein `AS_TO_AUDI` in einen `UDINT`-Wert (AUDI) konvertiert und an den ISOBUS Virtual Terminal-Ausgang `OutputNumber_N1` gesendet werden.
7. Testen Sie das Verhalten der Ablaufsteuerung in der Simulation (FORTE) oder auf der realen Hardware:
   - Läuft die Kette ohne Benutzereingriff vollautomatisch durch?
   - Lässt sich die Kette durch Tastendrücke beschleunigen?

2.1.2.296 Uebung_037_AX: Ablaufsteuerung 4-Kanal (Event & Zeit) ENDLOS (Adapter Version)

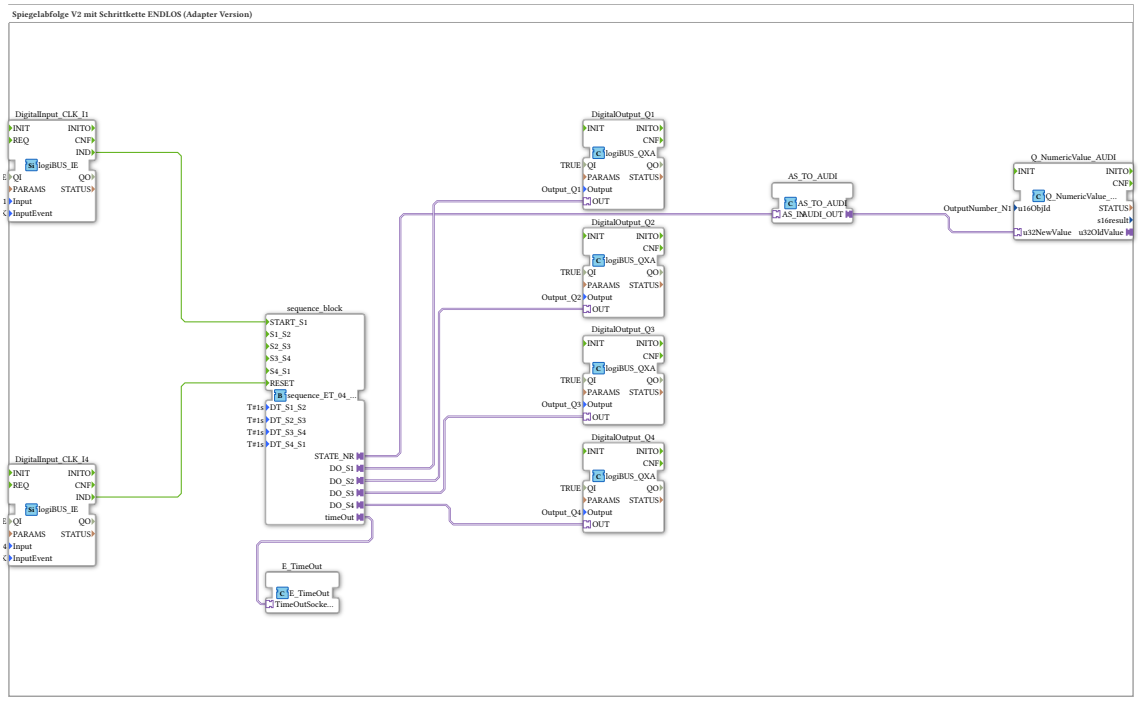
Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_037_AX`. Wir bauen eine kombinierte Event- und Zeit-Schrittkette mit AX-Adaptertechnologie.

Ziel der Übung

Realisierung einer Abfolge von 4 Ausgängen (Q1, Q2, Q3, Q4), gestartet und zurückgesetzt über Ereignisse (Taster), wobei die einzelnen Schritte zusätzlich über feste Zeiten (`DT_Sx_Sy`, je `T#1s`) weitergeschaltet werden - eine Kombination aus Start/Reset per Taster und automatischem Zeittakt zwischen den Schritten.

Beschreibung und Komponenten

Die Subapplikation `Uebung_037_AX` nutzt den kombinierten Sequenzer `sequence_ET_04_loop_AX`.



- **`sequence_ET_04_loop_AX`**: Startet auf Ereignis (`I1`), schaltet die 4 Schritte danach zeitgesteuert weiter.
- **`Q_NumericValue_AUDI`**: Zeigt die aktuelle Schrittnummer auf dem ISOBUS-Terminal an.

Funktionsweise

1. Start durch Taster `I1` -> Sequenz springt zu `S1`.
2. Jeder Ausgang (`Q1, Q2, Q3, Q4`) wird für die eingestellte Zeit aktiv geschaltet, bevor automatisch zum nächsten Schritt gewechselt wird.
3. Nach dem letzten Schritt kehrt die Sequenz automatisch zu S1 zurück (ENDLOS).
4. Reset durch Taster `I4` -> Alles aus.

2.1.2.297 Übung 037: Spiegelabfolge V2 mit Schrittkette ENDLOS

Thema: Ablaufsteuerungen in der Landtechnik

Situationsbeschreibung

Für ein System in der landwirtschaftlichen Automatisierung (z.B. eine Fassreinigung, eine Klappvorrichtung oder ein Gebläse) soll eine Ablaufsteuerung implementiert werden. Es werden 4 Stufen oder Aktoren nacheinander geschaltet.

Die Steuerung soll über den Baustein `sequence_ET_04_loop_AX` realisiert werden.

Funktionsbeschreibung der Ablaufsteuerung

- **Start:** Durch Betätigen des Tasters `I1` (`START_S1`).
- **Phasen:** 4 aufeinanderfolgende Ausgangsschritte (Ausgänge `Q1` bis `Q4`).
- **Übergänge:** Die Schrittübergänge sind eine Kombination aus Event und Zeitsteuerung.
- **Reset:** Zu jedem Zeitpunkt kann die Schrittkette über einen Reset (z.B. Taster `I4` oder Softkey) in den Ausgangszustand (Schritt 0) zurückgesetzt werden.

Arbeitsauftrag

1. Öffnen Sie das 4diac-Projekt und legen Sie die SubApp `Uebung_037_AX` bzw. `Uebung_037` an.
2. Platzieren Sie den Sequenzer-Baustein `sequence_ET_04_loop_AX` und konfigurieren Sie die Parameter.
3. Verbinden Sie den Start-Taster `I1` und den Reset-Taster `I4` mit den entsprechenden Eingängen.
4. Schalten Sie die 4 Ausgänge auf die physischen Ausgänge `Output_Q1` bis `Output_Q4`.
5. Zur Visualisierung soll die Zustandsnummer `STATE_NR` über den Baustein `AS_TO_AUDI` (Konvertierung `AS` ⟶ `AUDI`) auf dem ISOBUS Virtual Terminal-Ausgang `OutputNumber_N1` angezeigt werden.

2.1.2.298 Uebung_038_AX: Lauflicht 8 reine Zeitsteuerung



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_038_AX`. Wir bauen eine klassische Schrittkette (Sequencer).

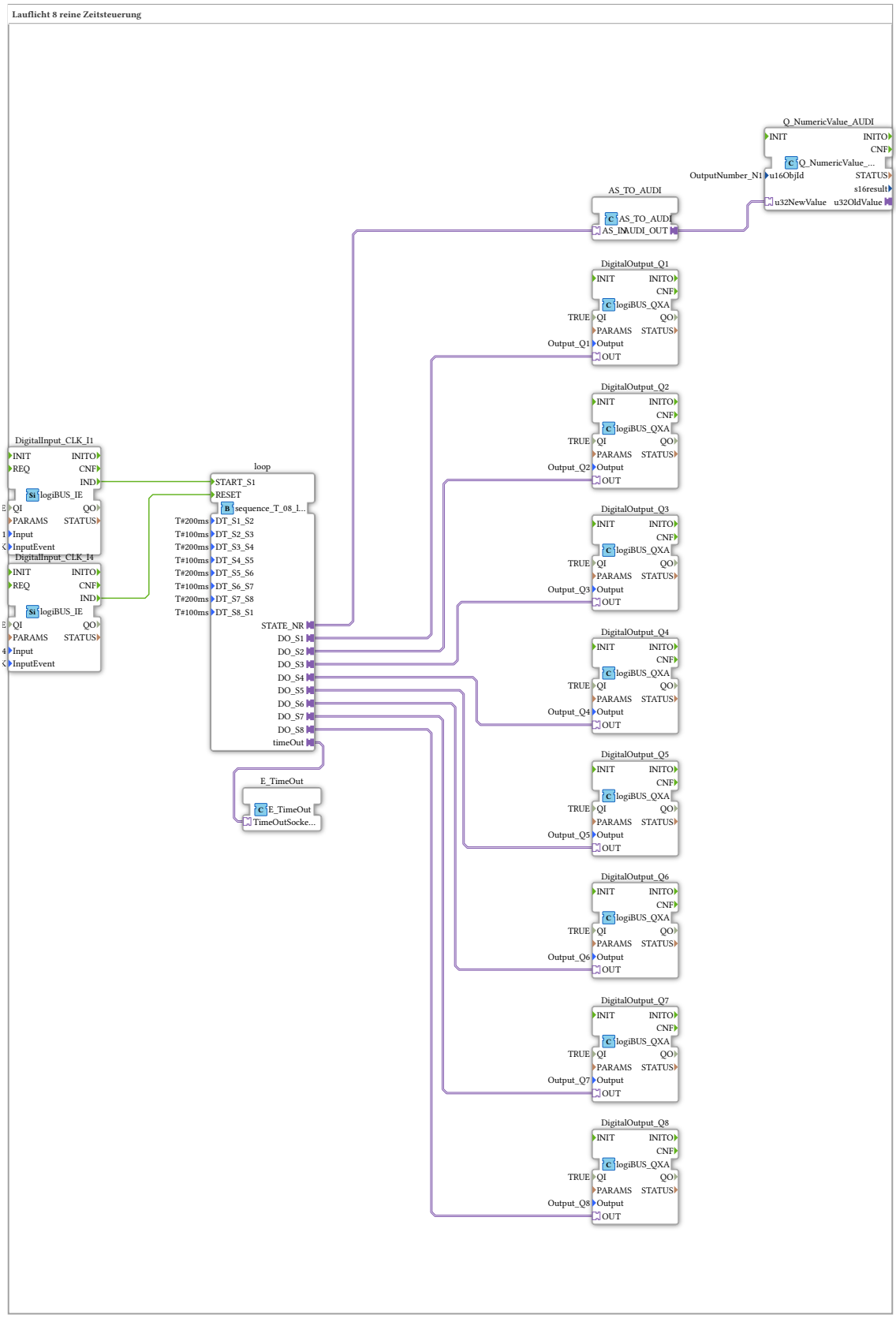
Ziel der Übung

Realisierung einer automatischen Abfolge von 8 Schritten.

Beschreibung und Komponenten

Die Subapplikation `Uebung_038_AX.SUB` verwendet einen Sequenzer-Baustein, um 8 Ausgänge nacheinander zu schalten.

Funktionsbausteine (FBs)



- **`sequence_T_08_loop_AX`**: Ein komplexer Baustein, der 8 Zustände (`S1` bis `S8`) verwaltet. Der Übergang zwischen den Zuständen erfolgt zeitgesteuert.
- **Parameter `DT_S1_S2` etc.**: Definieren die Verweildauer in jedem Schritt (z.B. 200ms oder 100ms).

- **`Q_NumericValue`**: Zeigt die aktuelle Schrittnummer auf dem ISOBUS-Terminal an.
- **`E_TimeOut`**: Überwacht die Sequenz (Watchdog).

Funktionsweise

1. Start durch Taster `I1` -> Sequenz springt zu `S1`.
2. Ausgang `DO_S1` wird aktiv -> `Q1` leuchtet.
3. Nach `T#200ms` wechselt die Sequenz automatisch zu `S2`.
4. `DO_S1` geht aus, `DO_S2` geht an -> `Q2` leuchtet.
5. ... das geht so weiter bis `S8`.
6. Nach `S8` springt sie wieder zu `S1` (Loop).
7. Reset durch Taster `I4` -> Alles aus.

Anwendungsbeispiel

Werbe-Beleuchtung oder **Prozess-Steuerung**: Zuerst Wasser einlassen, dann heizen, dann waschen, dann abpumpen.

2.1.2.299 Uebung_039_AX: Spiegelabfolge V2 mit Schrittkette

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_039_AX`. Diese Übung ist speziell auf die Ansteuerung von hydraulischen oder pneumatischen Wegeventilen zugeschnitten und nutzt durchgängig die AX-Adaptertechnologie.

Ziel der Übung

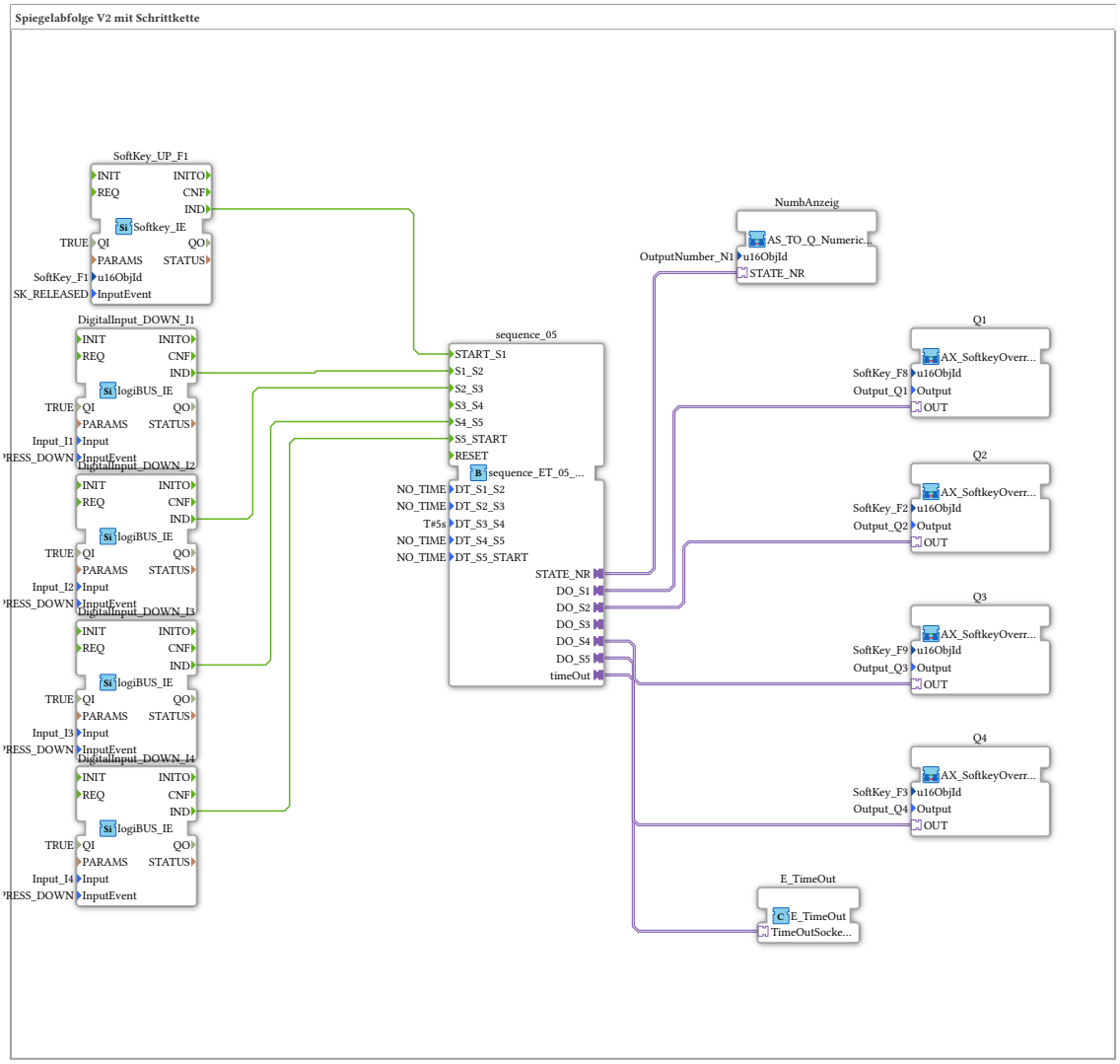
Realisierung einer komplexen Spiegelabfolge. Im Gegensatz zu einfachen Zylindern müssen bei Wegeventilen oft Zustände gehalten werden (Mittelstellung gesperrt), was eine präzise zeitliche und ereignisbasierte Steuerung der Spulen erfordert.

Beschreibung und Komponenten

Die Subapplikation `Uebung_039_AX.SUB` nutzt einen AX-optimierten 5-Schritt-Sequenzer (`sequence_ET_05_AX`). Die Ansteuerung der Hardware erfolgt über typisierte AX-Sub-Applikationen (`Uebung_039_sub_Outputs_AX`), die den jeweiligen Ventilzustand auch visuell auf dem ISOBUS-Terminal durch Farbumschlag der zugehörigen Softkeys rückmelden.

Funktionsweise

Die Kette wird manuell durch physische Taster (`I1` bis `I4`) gesteuert, wobei ein zentraler Zeitschritt (5s bei `DT_S3_S4`) eine automatische Sicherheits- oder Wartephase einfügt. Dies zeigt die Kombination aus freier Bedienbarkeit und erzwungenen Prozesszeiten.



Der Einsatz von **AX-Adaptern** zwischen dem Sequenzer und den Ausgangs-Sub-Apps vereinfacht die Verdrahtung erheblich, da Status-Events und Schaltzustände in einer einzigen Verbindung übertragen werden.

🛠️ Zugehörige Übungen

- AX_SoftkeyOverride_TO_QXA_BG
- Uebung_039_sub_NumbAnzeig_AX

2.1.2.300 Uebung_039_sub_NumbAnzeig_AS

Hinweis

Diese Übung existiert in den aktuellen Trainingsquellen nicht mehr (entfernt, zusammengelegt oder umbenannt). Diese Seite bleibt nur als Platzhalter für bestehende Verweise erhalten. Die aktuellen Übungen stehen in der Übungsübersicht.

2.1.2.301 Uebung_039_sub_NumbAnzeig_AX

Hinweis

Diese Übung existiert in den aktuellen Trainingsquellen nicht mehr (entfernt, zusammengelegt oder umbenannt). Diese Seite bleibt nur als Platzhalter für bestehende Verweise erhalten. Die aktuellen Übungen stehen in der Übungsübersicht.

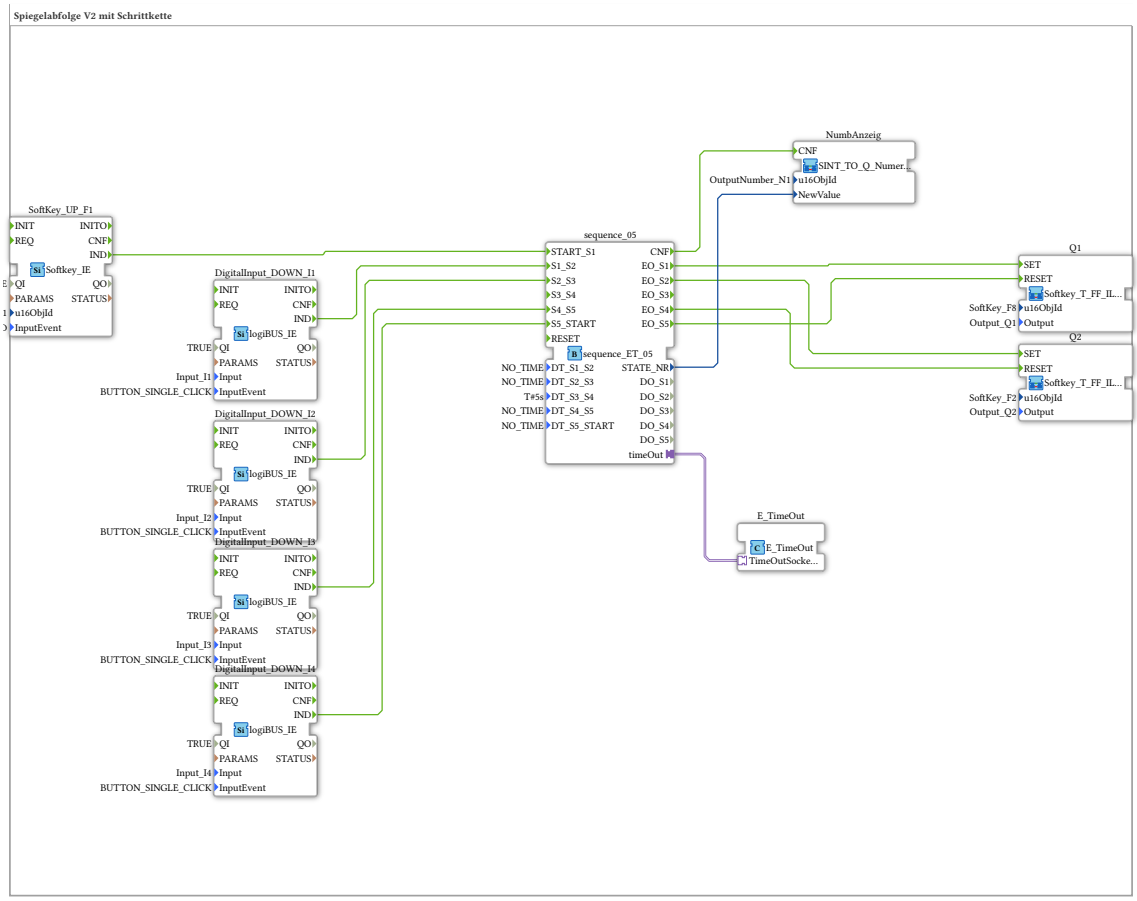
2.1.2.302 Uebung_039_sub_Outputs_AX

Hinweis

Diese Übung existiert nicht mehr als eigene Übung. Der Baustein wurde in die Bibliothek verschoben und heißt jetzt AX_SoftkeyOverride_TO_QXA_BG. Beschreibung, Schnittstelle und Netzwerk stehen dort.

2.1.2.303 Uebung_039a_AX: Spiegelabfolge V2 mit Schrittkette

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_039a_AX (Spiegelabfolge V2 mit Schrittkette).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Spiegelabfolge V2 mit Schrittkette**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_039a_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **E_TimeOut**: Instanz des Typs iec61499::events::E_TimeOut.
- **DigitalInput_DOWN_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **DigitalInput_DOWN_I2**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I2
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **DigitalInput_DOWN_I3**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE

  - Parameter Input = Input_I3
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **DigitalInput_DOWN_I4**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I4
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **sequence_05**: Instanz des Typs logiBUS::utils::sequence::combi::sequence_ET_05.
  - Parameter DT_S1_S2 = NO_TIME
  - Parameter DT_S2_S3 = NO_TIME
  - Parameter DT_S3_S4 = T#5s
  - Parameter DT_S4_S5 = NO_TIME
  - Parameter DT_S5_START = NO_TIME
- **SoftKey_UP_F1**: Instanz des Typs isobus::UT::io::Softkey::Softkey_IE.
  - Parameter QI = TRUE
  - Parameter u16ObjId = SoftKey_F1
  - Parameter InputEvent = SK_RELEASED

Verbindungen und Schnittstellen

Adapterverbindungen:

- sequence_05.timeOut -> E_TimeOut.TimeOutSocket

Ereignisverbindungen:

- DigitalInput_DOWN_I1.IND -> sequence_05.S1_S2
- DigitalInput_DOWN_I2.IND -> sequence_05.S2_S3
- DigitalInput_DOWN_I3.IND -> sequence_05.S4_S5
- DigitalInput_DOWN_I4.IND -> sequence_05.S5_START
- sequence_05.CNF -> NumbAnzeig.CNF
- SoftKey_UP_F1.IND -> sequence_05.START_S1
- sequence_05.EO_S1 -> Q1.SET
- sequence_05.EO_S2 -> Q2.SET
- sequence_05.EO_S4 -> Q2.RESET
- sequence_05.EO_S5 -> Q1.RESET

Datenverbindungen:

- sequence_05.STATE_NR -> NumbAnzeig.NewValue

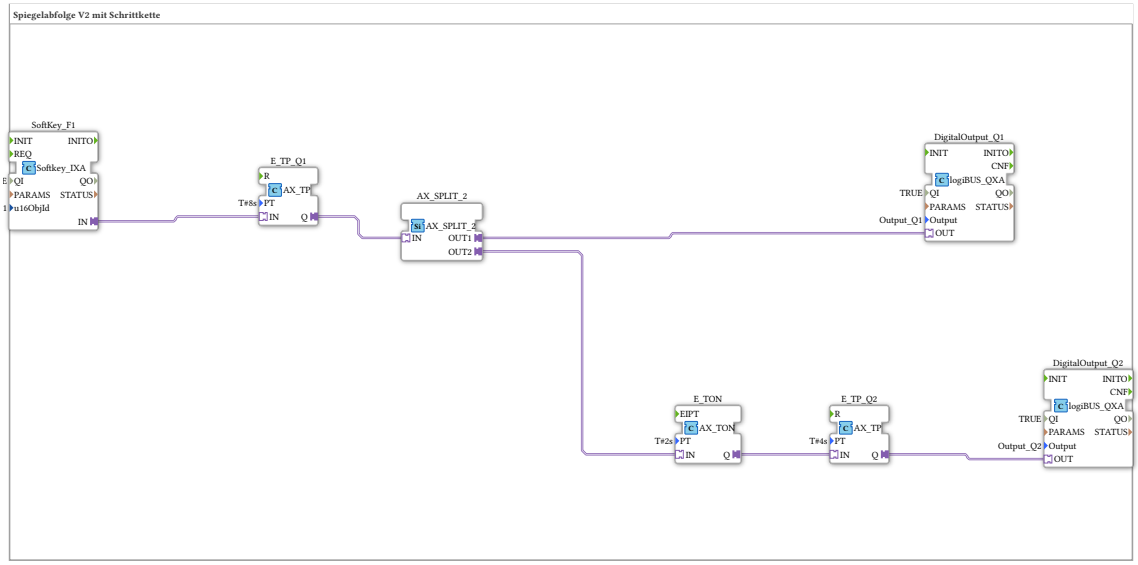
Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_039a_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.304 Uebung_039b_AX: Spiegelabfolge V2 mit Schrittkette



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung realisiert eine Spiegelabfolge (V2) als Schrittkette aus adapterbasierten Zeitgliedern: Ein Druck auf die Softkey-Taste `SoftKey_F1` startet einen 8 Sekunden langen Impuls auf `Q1`. Mit dem Impuls läuft zugleich eine Einschaltverzögerung von 2 Sekunden an, nach deren Ablauf ein 4 Sekunden langer Impuls auf `Q2` ausgelöst wird.

Verwendete Funktionsbausteine (FBs)

- **SoftKey_F1**: Softkey-Eingang des ISOBUS-Terminals
  - **Typ**: `isobus::UT::io::Softkey::Softkey_IXA`
  - **Parameter**: `QI = TRUE`, `u16ObjId = SoftKey_F1`
  - **Erklärung**: Liefert den Zustand der Softkey-Taste `SoftKey_F1` als AX-Adapterausgang `IN`.
- **E_TP_Q1**: Impulsgeber für `Q1`
  - **Typ**: `adapter::events::unidirectional::timers::AX_TP`
  - **Parameter**: `PT = T#8s`
  - **Erklärung**: Erzeugt aus dem Tastensignal einen Impuls der Länge 8 Sekunden am Adapterausgang `Q`.
- **AX_SPLIT_2**: Adapter-Signalverteiler
  - **Typ**: `adapter::events::unidirectional::AX_SPLIT_2`
  - **Parameter**: keine
  - **Erklärung**: Verteilt den Impuls von `E_TP_Q1.Q` auf zwei Ziele: `OUT1` (Ausgang `Q1`) und `OUT2` (Einschaltverzögerung).
- **E_TON**: Einschaltverzögerung
  - **Typ**: `adapter::events::unidirectional::timers::AX_TON`
  - **Parameter**: `PT = T#2s`
  - **Erklärung**: Verzögert das Einschalten des Signals aus `AX_SPLIT_2.OUT2` um 2 Sekunden.
- **E_TP_Q2**: Impulsgeber für `Q2`
  - **Typ**: `adapter::events::unidirectional::timers::AX_TP`
  - **Parameter**: `PT = T#4s`
  - **Erklärung**: Erzeugt aus dem verzögerten Signal von `E_TON.Q` einen Impuls der Länge 4 Sekunden.
- **DigitalOutput_Q1**: logiBUS Digitalausgang
  - **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
  - **Parameter**: `QI = TRUE`, `Output = Output_Q1`
  - **Erklärung**: Schaltet den physischen Ausgang `Q1` entsprechend dem 8-Sekunden-Impuls.
- **DigitalOutput_Q2**: logiBUS Digitalausgang
  - **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
  - **Parameter**: `QI = TRUE`, `Output = Output_Q2`
  - **Erklärung**: Schaltet den physischen Ausgang `Q2` entsprechend dem 4-Sekunden-Impuls.

Programmablauf und Verbindungen

1. `SoftKey_F1.IN` ⟶ `E_TP_Q1.IN` (AdapterConnection): Die Softkey-Taste startet den 8-Sekunden-Impuls.
2. `E_TP_Q1.Q` ⟶ `AX_SPLIT_2.IN` (AdapterConnection): Der Impuls wird auf zwei Ziele verteilt.
3. `AX_SPLIT_2.OUT1` ⟶ `DigitalOutput_Q1.OUT` (AdapterConnection): `Q1` folgt dem 8-Sekunden-Impuls.
4. `AX_SPLIT_2.OUT2` ⟶ `E_TON.IN` (AdapterConnection): Der Impuls startet die 2-Sekunden-Einschaltverzögerung.
5. `E_TON.Q` ⟶ `E_TP_Q2.IN` (AdapterConnection): Das verzögerte Signal startet den 4-Sekunden-Impuls.
6. `E_TP_Q2.Q` ⟶ `DigitalOutput_Q2.OUT` (AdapterConnection): `Q2` folgt dem 4-Sekunden-Impuls.

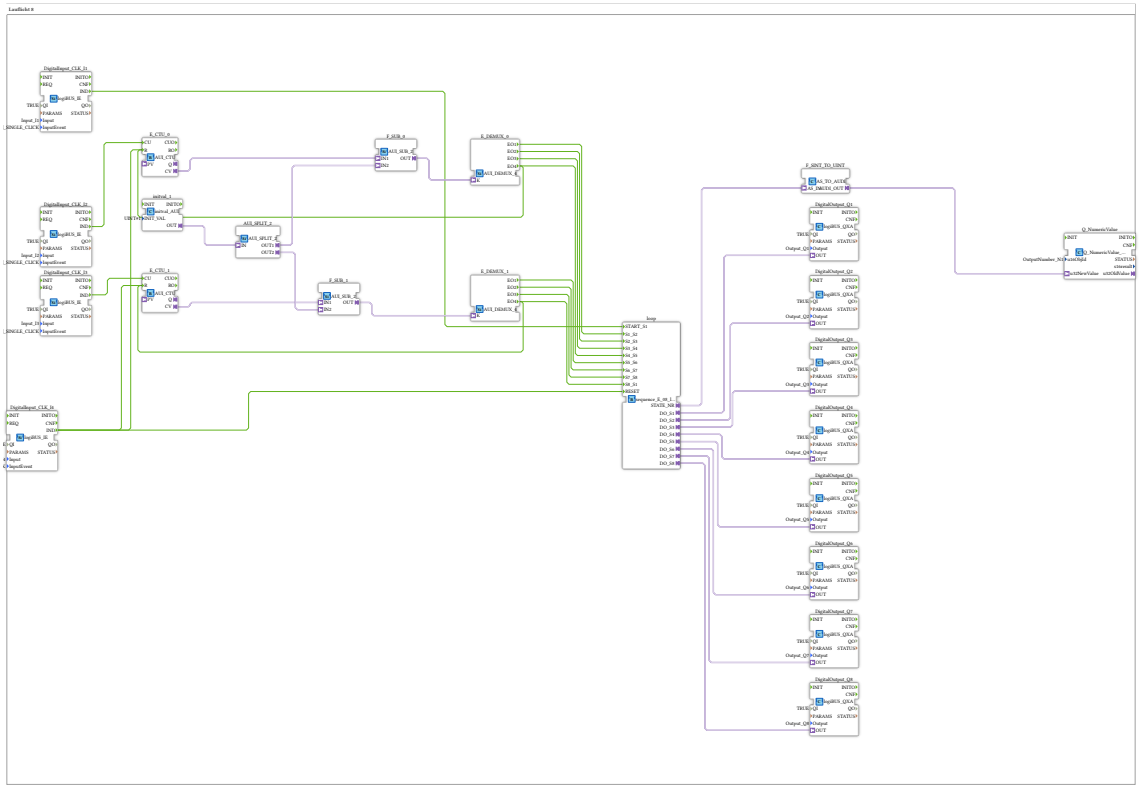
Zusammenfassung

Die Übung 039b_AX bildet eine zeitgesteuerte Schrittkette aus den Bausteinen `AX_TP` und `AX_TON`: Ein Tastendruck löst `Q1` für 8 Sekunden aus und startet über die Einschaltverzögerung von 2 Sekunden einen Impuls von 4 Sekunden auf `Q2`. `AX_SPLIT_2` verteilt dabei den Impuls ohne zusätzliche Ereignisverbindungen auf Ausgang und Folgeschritt.

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

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

2.1.2.305 Uebung_040_2_AX: Lauflicht 8



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung ist ein Lauflicht mit acht Ausgängen (`Q1` bis `Q8`), gesteuert durch den Sequenzbaustein `sequence_E_08_loop_AX` (Instanz `loop`). `I1` startet die Sequenz in Schritt 1, `I2` und `I3` schalten die Schritte weiter, `I4` setzt zurück. Die Schrittweiterschaltung läuft über zwei Zähler (`E_CTU_0`, `E_CTU_1`), die jeweils über einen Subtrahierer (Wert aus `initval_1` = 1) und einen Demultiplexer (`AUI_DEMUX_4`) vier Übergangs-Events der Sequenz auslösen. Die aktuelle Schrittnummer wird an der Anzeige `OutputNumber_N1` ausgegeben.

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_CLK_I1**, **DigitalInput_CLK_I2**, **DigitalInput_CLK_I3**, **DigitalInput_CLK_I4**: Taster mit Klick-Erkennung
  - **Typ**: `logiBUS::io::DI::logiBUS_IE`
  - **Parameter**: `QI = TRUE`, `Input = Input_I1` / `Input_I2` / `Input_I3` / `Input_I4`, jeweils `InputEvent = BUTTON_SINGLE_CLICK`
  - **Erklärung**: Melden einen Einzelklick auf `I1` bis `I4` als Event `IND`.
- **loop**: Lauflicht-Sequenz
  - **Typ**: `logiBUS::utils::sequence::event::sequence_E_08_loop_AX`
  - **Parameter**: keine
  - **Erklärung**: Acht Schritte als Ring. Die Eingänge `START_S1`, `S1_S2` bis `S8_S1` und `RESET` lösen die Übergänge aus; die Adapter `DO_S1` bis `DO_S8` zeigen den aktiven Schritt, `STATE_NR` liefert die Schrittnummer.
- **DigitalOutput_Q1** bis **DigitalOutput_Q8**: logiBUS Digitalausgänge
  - **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
  - **Parameter**: `QI = TRUE`, `Output = Output_Q1` bis `Output_Q8`
  - **Erklärung**: Jeder Ausgang `Qn` wird von `loop.DO_Sn` angesteuert.
- **E_CTU_0**, **E_CTU_1**: Zähler
  - **Typ**: `adapter::events::unidirectional::AUI_CTU`
  - **Parameter**: keine
  - **Erklärung**: `E_CTU_0` zählt die Klicks auf `I2`, `E_CTU_1` die Klicks auf `I3` (Eingang `CU`); der Zählwert `CV` geht an den Subtrahierer.
- **initval_1**: Startwert 1
  - **Typ**: `adapter::types::unidirectional::AUI::initval::initval_AUI`
  - **Parameter**: `INIT_VAL = 1`
  - **Erklärung**: Liefert den Wert 1 an `AUI_SPLIT_2`.
- **AUI_SPLIT_2**: Verteilung des Werts
  - **Typ**: `adapter::events::unidirectional::AUI_SPLIT_2`
  - **Parameter**: keine
  - **Erklärung**: `OUT1` geht an `F_SUB_0.IN2`, `OUT2` an `F_SUB_1.IN2`.
- **F_SUB_0**, **F_SUB_1**: Subtrahierer
  - **Typ**: `adapter::iec61131::arithmetic::AUI_SUB_2`
  - **Parameter**: keine
  - **Erklärung**: Bilden `IN1 - IN2` aus dem Zählwert `CV` (`IN1`) und dem Wert aus `initval_1` (`IN2`); das Ergebnis `OUT` geht an den Demultiplexer.
- **E_DEMUX_0**, **E_DEMUX_1**: Demultiplexer
  - **Typ**: `adapter::events::unidirectional::AUI_DEMUX_4`
  - **Parameter**: keine
  - **Erklärung**: Der Wert am Eingang `K` wählt eines der vier Events `EO1` bis `EO4`. `E_DEMUX_0` löst die Übergänge `S1_S2` bis `S4_S5` aus, `E_DEMUX_1` die Übergänge `S5_S6` bis `S8_S1`.
- **F_SINT_TO_UINT**: Konvertierung der Schrittnummer
  - **Typ**: `adapter::conversion::unidirectional::AS_TO_AUDI`
  - **Parameter**: keine
  - **Erklärung**: Wandelt `loop.STATE_NR` (`AS_IN`) in einen `AUDI`-Wert.
- **Q_NumericValue**: numerische Anzeige
  - **Typ**: `isobus::UT::Q::Q_NumericValue_AUDI`
  - **Parameter**: `u16ObjId = OutputNumber_N1`
  - **Erklärung**: Zeigt den Wert an `u32NewValue` an `OutputNumber_N1` an.

Programmablauf und Verbindungen

1. `DigitalInput_CLK_I1.IND` ⟶ `loop.START_S1` (EventConnection): `I1` startet die Sequenz in Schritt 1.
2. `DigitalInput_CLK_I2.IND` ⟶ `E_CTU_0.CU` und `DigitalInput_CLK_I3.IND` ⟶ `E_CTU_1.CU` (EventConnections): Jeder Klick erhöht den jeweiligen Zähler.
3. `E_CTU_0.CV` ⟶ `F_SUB_0.IN1` und `AUI_SPLIT_2.OUT1` ⟶ `F_SUB_0.IN2`, `F_SUB_0.OUT` ⟶ `E_DEMUX_0.K` (AdapterConnections): Zählwert minus Startwert wählt den Demultiplexer-Ausgang. Dasselbe gilt für `E_CTU_1`, `F_SUB_1` (`AUI_SPLIT_2.OUT2`) und `E_DEMUX_1`; `initval_1.OUT` ⟶ `AUI_SPLIT_2.IN` speist beide.

4. `E_DEMUX_0.EO1` bis `EO4` ⟶ `loop.S1_S2`, `S2_S3`, `S3_S4`, `S4_S5` sowie `E_DEMUX_1.EO1` bis `EO4` ⟶ `loop.S5_S6`, `S6_S7`, `S7_S8`, `S8_S1` (EventConnections): Die Demultiplexer lösen die acht Schrittübergänge aus.
5. `DigitalInput_CLK_I4.IND` ⟶ `loop.RESET`, `E_CTU_0.R` und `E_CTU_1.R` (EventConnections): `I4` setzt Sequenz und beide Zähler zurück. Zusätzlich setzt `E_DEMUX_0.EO4` den Zähler `E_CTU_0` und `E_DEMUX_1.EO4` den Zähler `E_CTU_1` zurück.
6. `loop.DO_S1` bis `loop.DO_S8` ⟶ `DigitalOutput_Q1.OUT` bis `DigitalOutput_Q8.OUT` (AdapterConnections): Der aktive Schritt steuert den zugehörigen Ausgang.
7. `loop.STATE_NR` ⟶ `F_SINT_TO_UINT.AS_IN`, `F_SINT_TO_UINT.AUDI_OUT` ⟶ `Q_NumericValue.u32NewValue` (AdapterConnections): Die Schrittnummer wird an `OutputNumber_N1` angezeigt.

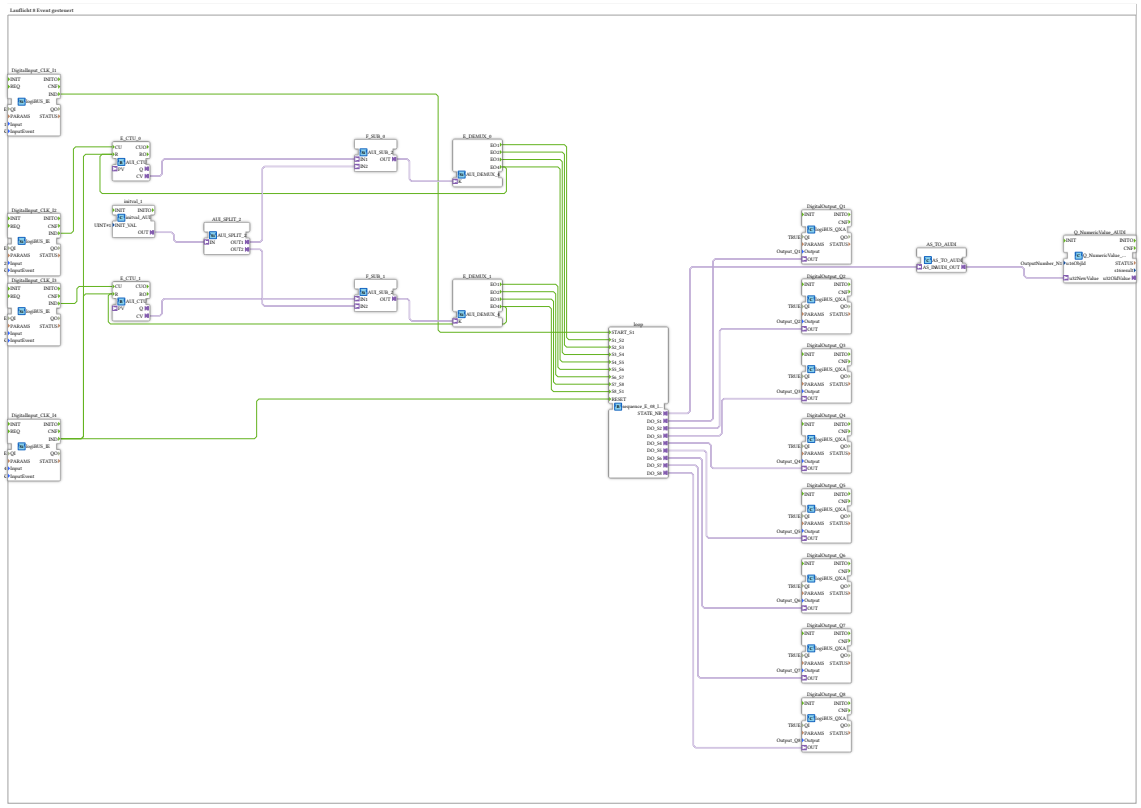
Zusammenfassung

Die Übung 040_2_AX baut ein Lauflicht mit acht Schritten aus dem Sequenzbaustein `sequence_E_08_loop_AX` auf. Die Schrittübergänge werden nicht automatisch, sondern über zwei Zähler mit Subtrahierer und Demultiplexer durch Klicks auf `I2` und `I3` ausgelöst; `I1` startet, `I4` setzt zurück. Alle Verbindungen zwischen den Bausteinen laufen über Adapter.

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

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

2.1.2.306 Uebung_040_AX: Lauflicht 8 Event gesteuert



• – ○ • – ○ • – ○ •

Einleitung

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_040_AX`. Im Gegensatz zu Übung 038 schaltet diese Schrittkette nicht automatisch weiter, sondern wartet auf Ereignisse. Ziel ist das manuelle Weiterschalten einer Schrittkette mit 8 Schritten.

Die Subapplikation `Uebung_040_AX.SUB` nutzt `sequence_E_08_loop_AX`. Hier sind die Eingänge für die Transitionen (`S1_S2`, `S2_S3`, …) als Event-Inputs herausgeführt. Um nicht 8 Taster zu benötigen, wurde eine Logik mit Zählern (`AUI_CTU`) und Demultiplexern (`AUI_DEMUX_4`) gebaut.

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_CLK_I1**: Start-Taster
  - **Typ**: `logiBUS::io::DI::logiBUS_IE`
  - **Parameter**: `QI = TRUE`, `Input = Input_I1`, `InputEvent = BUTTON_SINGLE_CLICK`
  - **Erklärung**: Einzelklick auf `I1` ⟶ `loop.START_S1`.
- **DigitalInput_CLK_I2**: Weiterschalten der Schritte 1-4
  - **Typ**: `logiBUS::io::DI::logiBUS_IE`
  - **Parameter**: `QI = TRUE`, `Input = Input_I2`, `InputEvent = BUTTON_SINGLE_CLICK`
  - **Erklärung**: Einzelklick auf `I2` ⟶ `E_CTU_0.CU` (Zähler hochzählen).
- **DigitalInput_CLK_I3**: Weiterschalten der Schritte 5-8
  - **Typ**: `logiBUS::io::DI::logiBUS_IE`
  - **Parameter**: `QI = TRUE`, `Input = Input_I3`, `InputEvent = BUTTON_SINGLE_CLICK`
  - **Erklärung**: Einzelklick auf `I3` ⟶ `E_CTU_1.CU`.
- **DigitalInput_CLK_I4**: Reset-Taster
  - **Typ**: `logiBUS::io::DI::logiBUS_IE`
  - **Parameter**: `QI = TRUE`, `Input = Input_I4`, `InputEvent = BUTTON_SINGLE_CLICK`
  - **Erklärung**: Einzelklick auf `I4` ⟶ `loop.RESET`, `E_CTU_0.R` und `E_CTU_1.R`.
- **loop**: 8-Schritt-Schrittkette (Ereignis-gesteuert, im Kreis)
  - **Typ**: `logiBUS::utils::sequence::event::sequence_E_08_loop_AX`
  - **Parameter**: keine
  - **Erklärung**: Zustände `START` und `State_01` bis `State_08`; Übergänge über die Event-Eingänge `START_S1`, `S1_S2` … `S7_S8`, `S8_S1` sowie `RESET` (aus jedem Zustand zurück nach `START`). Ausgänge als Adapter: `STATE_NR` (AS) und `DO_S1` bis `DO_S8` (AX).
- **initval_1**: Startwert 1
  - **Typ**: `adapter::types::unidirectional::AUI::initval::initval_AUI`
  - **Parameter**: `INIT_VAL = 1`
  - **Erklärung**: Liefert beim Initialisieren den UINT-Wert 1 als Adapter `OUT`.
- **AUI_SPLIT_2**: Verteiler für den Startwert
  - **Typ**: `adapter::events::unidirectional::AUI_SPLIT_2`
  - **Parameter**: keine
  - **Erklärung**: Verteilt den Wert 1 von `initval_1` an `F_SUB_0.IN2` (`OUT1`) und `F_SUB_1.IN2` (`OUT2`).
- **E_CTU_0**, **E_CTU_1**: Ereigniszähler
  - **Typ**: `adapter::events::unidirectional::AUI_CTU`
  - **Parameter**: keine
  - **Erklärung**: Zählen bei `CU` den Zählerwert `CV` um 1 hoch, `R` setzt ihn auf 0 zurück. `E_CTU_0` wird von `I2`, `E_CTU_1` von `I3` hochgezählt.
- **F_SUB_0**, **F_SUB_1**: Subtraktion
  - **Typ**: `adapter::iec61131::arithmetic::AUI_SUB_2`
  - **Parameter**: keine
  - **Erklärung**: Bilden `CV - 1` (`IN1` = Zählerwert, `IN2` = Startwert 1), damit der erste Klick den Index 0 ergibt.
- **E_DEMUX_0**, **E_DEMUX_1**: Ereignis-Demultiplexer
  - **Typ**: `adapter::events::unidirectional::AUI_DEMUX_4`
  - **Parameter**: keine
  - **Erklärung**: Löst abhängig vom Index `K` den Ausgang `EO1` (K=0), `EO2` (K=1), `EO3` (K=2) oder `EO4` (K=3) aus.
- **AS_TO_AUDI**: Adapterkonverter SINT ⟶ UDINT
  - **Typ**: `adapter::conversion::unidirectional::AS_TO_AUDI`
  - **Parameter**: keine
  - **Erklärung**: Wandelt die Schrittnummer `loop.STATE_NR` (AS) in einen `AUDI`-Adapter.
- **Q_NumericValue_AUDI**: VT-Zahlenfeld

  - **Typ**: `isobus::UT::Q::Q_NumericValue_AUDI`
  - **Parameter**: `u16ObjId = OutputNumber_N1`
  - **Erklärung**: Zeigt die Schrittnummer im VT-Zahlenfeld `OutputNumber_N1`.
- **DigitalOutput_Q1** bis **DigitalOutput_Q8**: Digitalausgänge
  - **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
  - **Parameter**: `QI = TRUE`, `Output = Output_Q1` ... `Output_Q8`
  - **Erklärung**: `DigitalOutput_Qn` wird von `loop.DO_Sn` angesteuert.

Programmablauf und Verbindungen

1. `DigitalInput_CLK_I1.IND` ⟶ `loop.START_S1`: Start der Kette, Schritt 1 aktiv.
2. `DigitalInput_CLK_I2.IND` ⟶ `E_CTU_0.CU`; `E_CTU_0.CV` und der Wert 1 (`AUI_SPLIT_2.OUT1`) ⟶ `F_SUB_0` ⟶ `E_DEMUX_0.K`. `E_DEMUX_0.EO1` ⟶ `loop.S1_S2`, `EO2` ⟶ `S2_S3`, `EO3` ⟶ `S3_S4`, `EO4` ⟶ `S4_S5`.
3. `DigitalInput_CLK_I3.IND` ⟶ `E_CTU_1.CU`; über `F_SUB_1` ⟶ `E_DEMUX_1.K`. `E_DEMUX_1.EO1` ⟶ `loop.S5_S6`, `EO2` ⟶ `S6_S7`, `EO3` ⟶ `S7_S8`, `EO4` ⟶ `S8_S1`.
4. `E_DEMUX_0.EO4` ⟶ `E_CTU_0.R` und `E_DEMUX_1.EO4` ⟶ `E_CTU_1.R`: jeder Zähler setzt sich nach dem vierten Klick selbst zurück.
5. `DigitalInput_CLK_I4.IND` ⟶ `loop.RESET`, `E_CTU_0.R`, `E_CTU_1.R`: Reset von Kette und beiden Zählern.
6. `loop.DO_S1` ... `loop.DO_S8` ⟶ `DigitalOutput_Q1` ... `DigitalOutput_Q8`.
7. `loop.STATE_NR` ⟶ `AS_TO_AUDI.AS_IN`, `AS_TO_AUDI.AUDI_OUT` ⟶ `Q_NumericValue_AUDI.u32NewValue`: Anzeige der Schrittnummer.

Verhalten:

1. Start mit `I1` ⟶ Schritt 1 aktiv.
2. Druck auf `I2` ⟶ Zählerstand 1 ⟶ Index 0 ⟶ Demux `EO1` ⟶ Event an `S1_S2` ⟶ Wechsel zu Schritt 2.
3. Druck auf `I2` ⟶ Zählerstand 2 ⟶ Index 1 ⟶ Demux `EO2` ⟶ Event an `S2_S3` ⟶ Wechsel zu Schritt 3.
4. Der vierte Druck auf `I2` löst `S4_S5` aus (Schritt 5) und setzt den Zähler zurück. Danach wird mit `I3` analog durch die Schritte 5 bis 8 und zurück zu Schritt 1 (`S8_S1`) geschaltet.

Dies simuliert eine Maschine, bei der der Bediener jeden Schritt manuell freigeben muss („Schrittbetrieb“).

Zusammenfassung

Die Übung zeigt, wie mit zwei Zählern und zwei Demultiplexern aus nur zwei Tastern (`I2`, `I3`) acht Transitionen einer Ereignis-gesteuerten Schrittkette bedient werden können; `I1` startet, `I4` setzt zurück.

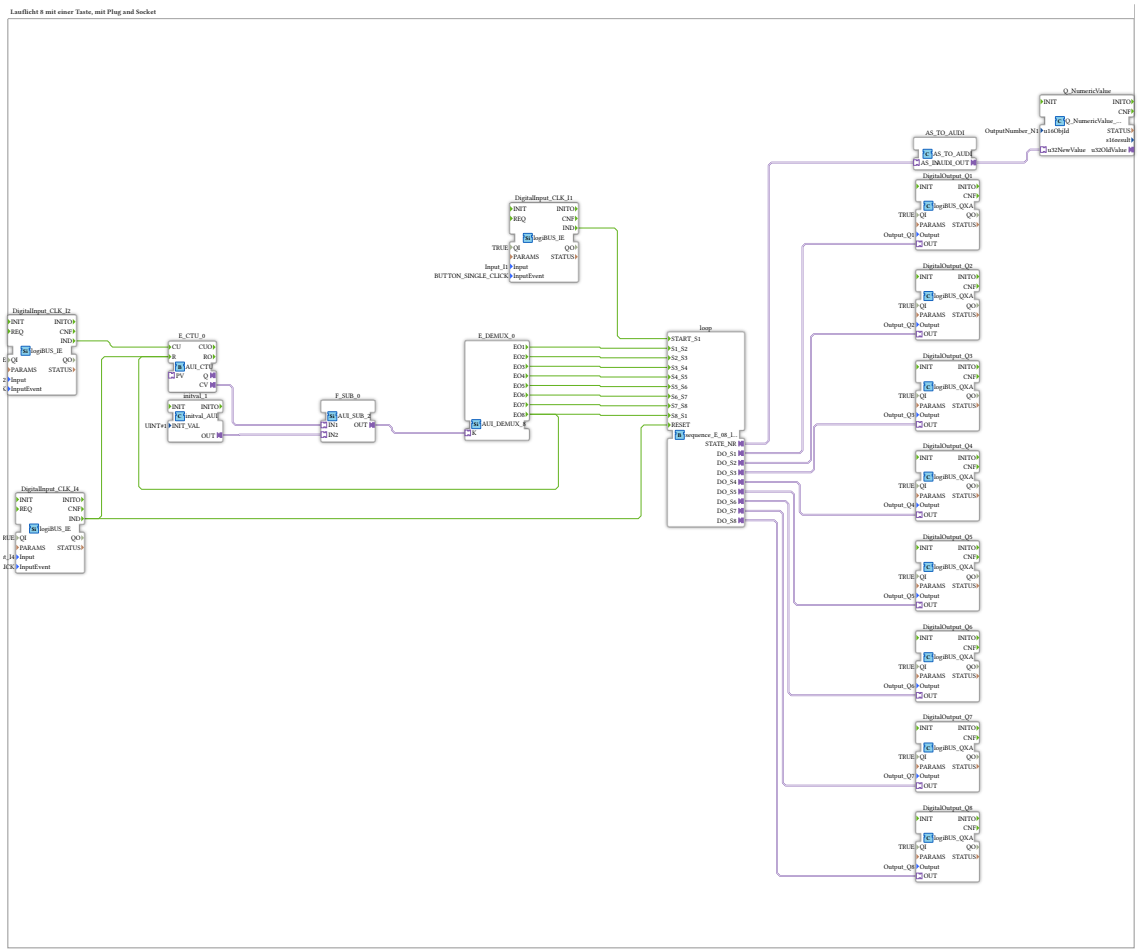
Anwendungsbeispiel - Inbetriebnahme oder Wartung: Der Techniker schaltet die Maschine Schritt für Schritt weiter, um zu prüfen, ob jeder Teilprozess korrekt funktioniert, bevor er auf Automatik umschaltet.

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

- 🌐 E_CTU Event Counter Baustein auf ms-muc-docs.de

2.1.2.307 Uebung_041_AX: Lauflicht 8 mit einer Taste, mit Plug and Socket

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_041_AX (Lauflicht 8 mit einer Taste, mit Plug and Socket).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Lauflicht 8 mit einer Taste, mit Plug and Socket**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_041_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
- **DigitalOutput_Q2**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q2
- **DigitalOutput_Q3**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q3
- **DigitalOutput_Q4**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q4
- **DigitalInput_CLK_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **DigitalInput_CLK_I4**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.

  - Parameter QI = TRUE
  - Parameter Input = Input_I4
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **DigitalOutput_Q5**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q5
- **DigitalOutput_Q6**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q6
- **DigitalOutput_Q7**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q7
- **DigitalOutput_Q8**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q8
- **Q_NumericValue**: Instanz des Typs isobus::UT::Q::Q_NumericValue_AUDI.
  - Parameter u16ObjId = OutputNumber_N1
- **AS_TO_AUDI**: Instanz des Typs adapter::conversion::unidirectional::AS_TO_AUDI.
- **loop**: Instanz des Typs logiBUS::utils::sequence::event::sequence_E_08_loop_AX.
- **E_CTU_0**: Instanz des Typs adapter::events::unidirectional::AUI_CTU.
- **F_SUB_0**: Instanz des Typs adapter::iec61131::arithmetic::AUI_SUB_2.
- **DigitalInput_CLK_I2**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I2
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **E_DEMUX_0**: Instanz des Typs adapter::events::unidirectional::AUI_DEMUX_8.
- **initval_1**: Instanz des Typs adapter::types::unidirectional::AUI::initval::initval_AUI.
  - Parameter INIT_VAL = 1

Verbindungen und Schnittstellen

Adapterverbindungen:

- loop.DO_S1 -> DigitalOutput_Q1.OUT
- loop.DO_S2 -> DigitalOutput_Q2.OUT
- loop.DO_S3 -> DigitalOutput_Q3.OUT
- loop.DO_S4 -> DigitalOutput_Q4.OUT
- loop.DO_S5 -> DigitalOutput_Q5.OUT
- loop.DO_S6 -> DigitalOutput_Q6.OUT
- loop.DO_S7 -> DigitalOutput_Q7.OUT
- loop.DO_S8 -> DigitalOutput_Q8.OUT
- loop.STATE_NR -> AS_TO_AUDI.AS_IN
- AS_TO_AUDI.AUDI_OUT -> Q_NumericValue.u32NewValue
- E_CTU_0.CV -> F_SUB_0.IN1
- initval_1.OUT -> F_SUB_0.IN2
- F_SUB_0.OUT -> E_DEMUX_0.K

Ereignisverbindungen:

- DigitalInput_CLK_I4.IND -> loop.RESET
- DigitalInput_CLK_I1.IND -> loop.START_S1
- DigitalInput_CLK_I2.IND -> E_CTU_0.CU
- DigitalInput_CLK_I4.IND -> E_CTU_0.R
- E_DEMUX_0.EO2 -> loop.S2_S3
- E_DEMUX_0.EO3 -> loop.S3_S4
- E_DEMUX_0.EO4 -> loop.S4_S5
- E_DEMUX_0.EO5 -> loop.S5_S6
- E_DEMUX_0.EO6 -> loop.S6_S7
- E_DEMUX_0.EO7 -> loop.S7_S8
- E_DEMUX_0.EO8 -> loop.S8_S1
- E_DEMUX_0.EO8 -> E_CTU_0.R
- E_DEMUX_0.EO1 -> loop.S1_S2

Programmablauf und Funktionsweise

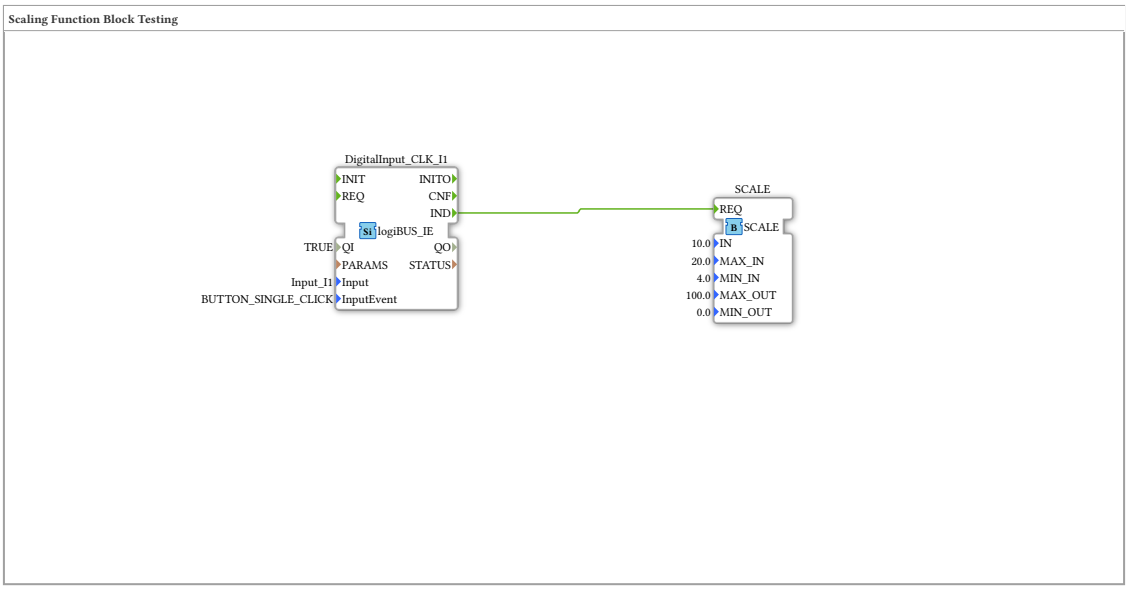
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_041_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.308 Uebung_042_AX: Scaling Function Block Testing

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_042_AX (Scaling Function Block Testing).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Scaling Function Block Testing**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_042_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_CLK_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **SCALE**: Instanz des Typs eclipse4diac::signalprocessing::SCALE.
  - Parameter IN = 10.0
  - Parameter MAX_IN = 20.0
  - Parameter MIN_IN = 4.0
  - Parameter MAX_OUT = 100.0
  - Parameter MIN_OUT = 0.0

Verbindungen und Schnittstellen

Ereignisverbindungen:

- DigitalInput_CLK_I1.IND -> SCALE.REQ

Programmablauf und Funktionsweise

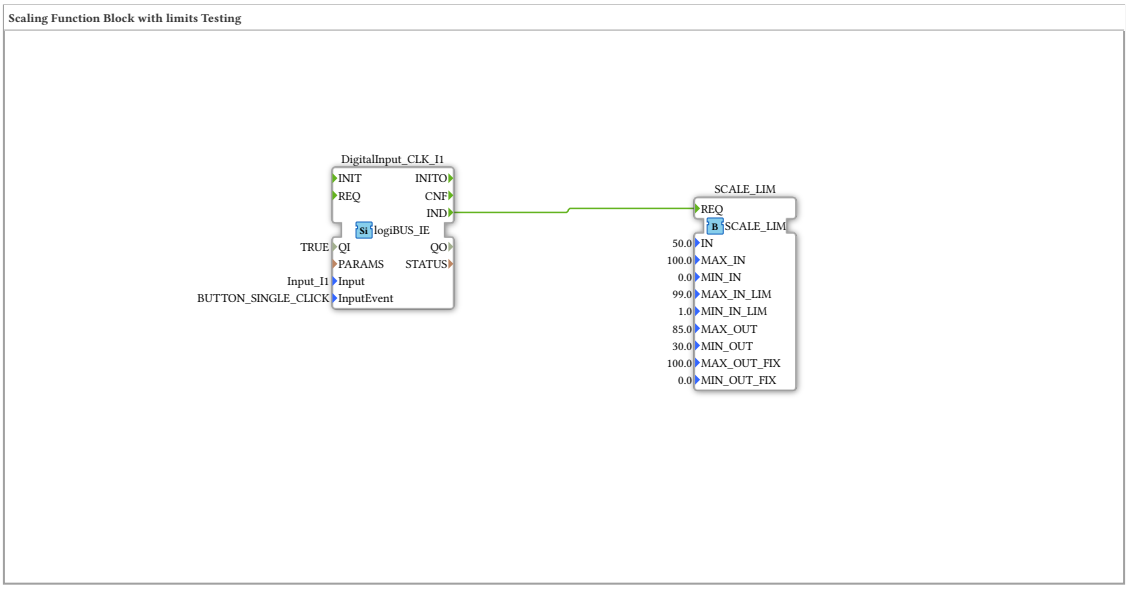
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_042_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.309 Uebung_043_AX: Scaling Function Block with limits Testing

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_043_AX (Scaling Function Block with limits Testing).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Scaling Function Block with limits Testing**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_043_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_CLK_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **SCALE_LIM**: Instanz des Typs eclipse4diac::signalprocessing::SCALE_LIM.
  - Parameter IN = 50.0
  - Parameter MAX_IN = 100.0
  - Parameter MIN_IN = 0.0
  - Parameter MAX_IN_LIM = 99.0
  - Parameter MIN_IN_LIM = 1.0
  - Parameter MAX_OUT = 85.0
  - Parameter MIN_OUT = 30.0
  - Parameter MAX_OUT_FIX = 100.0
  - Parameter MIN_OUT_FIX = 0.0

Verbindungen und Schnittstellen

Ereignisverbindungen:

- DigitalInput_CLK_I1.IND -> SCALE_LIM.REQ

Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_043_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.310 Uebung_044_AX: Ablaufsteuerung 5-Kanal (Event & Zeit) ENDLOS (Adapter Version)

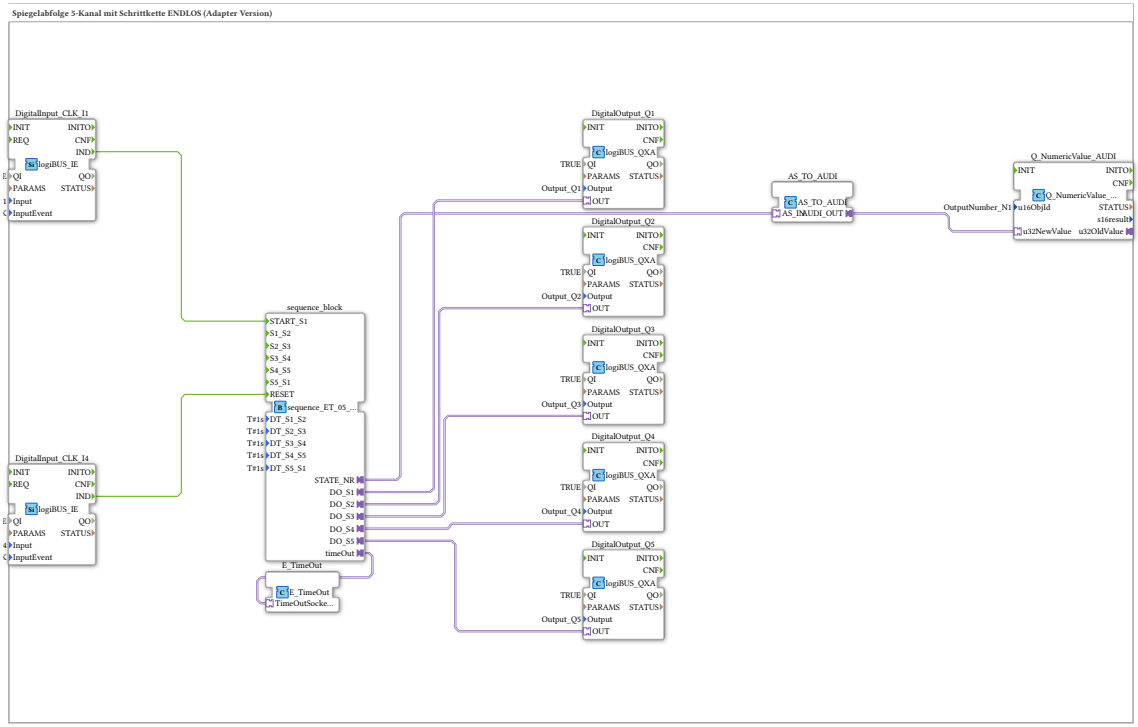
Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_044_AX`. Wir bauen eine kombinierte Event- und Zeit-Schrittkette mit AX-Adaptertechnologie.

Ziel der Übung

Realisierung einer Abfolge von 5 Ausgängen (Q1, Q2, Q3, Q4, Q5), gestartet und zurückgesetzt über Ereignisse (Taster), wobei die einzelnen Schritte zusätzlich über feste Zeiten (`DT_Sx_Sy`, je `T#1s`) weitergeschaltet werden - eine Kombination aus Start/Reset per Taster und automatischem Zeittakt zwischen den Schritten.

Beschreibung und Komponenten

Die Subapplikation `Uebung_044_AX` nutzt den kombinierten Sequenzer `sequence_ET_05_loop_AX`.



- **`sequence_ET_05_loop_AX`**: Startet auf Ereignis (`I1`), schaltet die 5 Schritte danach zeitgesteuert weiter.
- **`Q_NumericValue_AUDI`**: Zeigt die aktuelle Schrittnummer auf dem ISOBUS-Terminal an.

Funktionsweise

1. Start durch Taster `I1` -> Sequenz springt zu `S1`.
2. Jeder Ausgang (`Q1, Q2, Q3, Q4, Q5`) wird für die eingestellte Zeit aktiv geschaltet, bevor automatisch zum nächsten Schritt gewechselt wird.
3. Nach dem letzten Schritt kehrt die Sequenz automatisch zu S1 zurück (ENDLOS).
4. Reset durch Taster `I4` -> Alles aus.

2.1.2.311 Übung 044: Spiegelabfolge 5-Kanal mit Schrittkette ENDLOS

Thema: Ablaufsteuerungen in der Landtechnik

Situationsbeschreibung

Für ein System in der landwirtschaftlichen Automatisierung (z.B. eine Fassreinigung, eine Klappvorrichtung oder ein Gebläse) soll eine Ablaufsteuerung implementiert werden. Es werden 5 Stufen oder Aktoren nacheinander geschaltet.

Die Steuerung soll über den Baustein `sequence_ET_05_loop_AX` realisiert werden.

Funktionsbeschreibung der Ablaufsteuerung

- **Start:** Durch Betätigen des Tasters `I1` (`START_S1`).
- **Phasen:** 5 aufeinanderfolgende Ausgangsschritte (Ausgänge `Q1` bis `Q5`).
- **Übergänge:** Die Schrittübergänge sind eine Kombination aus Event und Zeitsteuerung.
- **Reset:** Zu jedem Zeitpunkt kann die Schrittkette über einen Reset (z.B. Taster `I4` oder Softkey) in den Ausgangszustand (Schritt 0) zurückgesetzt werden.

Arbeitsauftrag

1. Öffnen Sie das 4diac-Projekt und legen Sie die SubApp `Uebung_044_AX` bzw. `Uebung_044` an.
2. Platzieren Sie den Sequenzer-Baustein `sequence_ET_05_loop_AX` und konfigurieren Sie die Parameter.
3. Verbinden Sie den Start-Taster `I1` und den Reset-Taster `I4` mit den entsprechenden Eingängen.
4. Schalten Sie die 5 Ausgänge auf die physischen Ausgänge `Output_Q1` bis `Output_Q5`.
5. Zur Visualisierung soll die Zustandsnummer `STATE_NR` auf dem ISOBUS Virtual Terminal-Ausgang `OutputNumber_N1` angezeigt werden.

2.1.2.312 Uebung_045_AX: Ablaufsteuerung 8-Kanal (Event & Zeit) (Adapter Version)

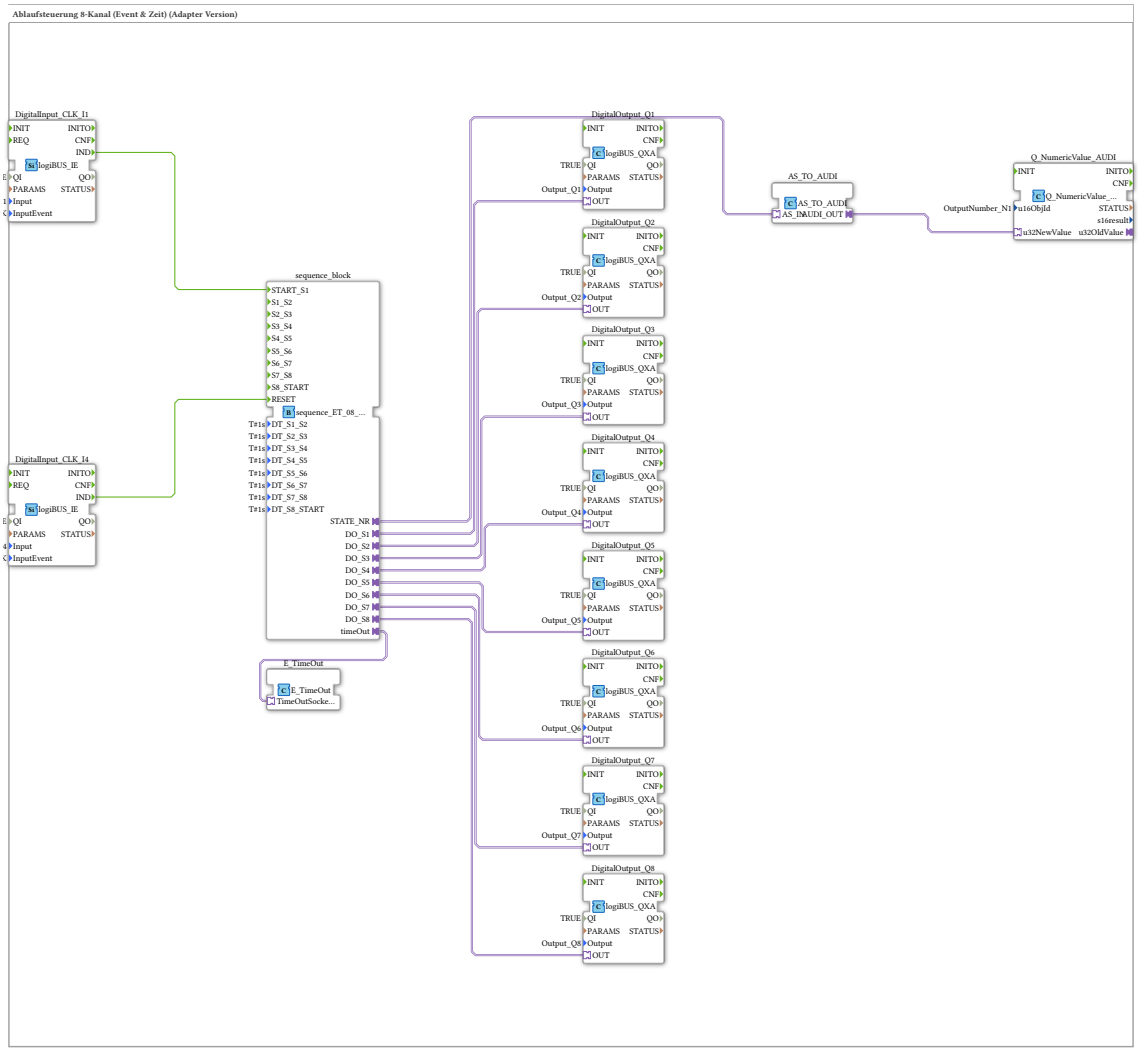
Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_045_AX`. Wir bauen eine kombinierte Event- und Zeit-Schrittkette mit AX-Adaptertechnologie.

Ziel der Übung

Realisierung einer Abfolge von 8 Ausgängen (Q1, Q2, Q3, Q4, Q5, Q6, Q7, Q8), gestartet und zurückgesetzt über Ereignisse (Taster), wobei die einzelnen Schritte zusätzlich über feste Zeiten (`DT_Sx_Sy`, je `T#1s`) weitergeschaltet werden - eine Kombination aus Start/Reset per Taster und automatischem Zeittakt zwischen den Schritten.

Beschreibung und Komponenten

Die Subapplikation `Uebung_045_AX` nutzt den kombinierten Sequenzer `sequence_ET_08_AX`.



- **`sequence_ET_08_AX`**: Startet auf Ereignis (`I1`), schaltet die 8 Schritte danach zeitgesteuert weiter.
- **`Q_NumericValue_AUDI`**: Zeigt die aktuelle Schrittnummer auf dem ISOBUS-Terminal an.

Funktionsweise

1. Start durch Taster `I1` -> Sequenz springt zu `S1`.
2. Jeder Ausgang (`Q1, Q2, Q3, Q4, Q5, Q6, Q7, Q8`) wird für die eingestellte Zeit aktiv geschaltet, bevor automatisch zum nächsten Schritt gewechselt wird.
3. Nach dem letzten Schritt endet die Sequenz.
4. Reset durch Taster `I4` -> Alles aus.

2.1.2.313 Übung 045: Ablaufsteuerung 8-Kanal (Event & Zeit)

Thema: Ablaufsteuerungen in der Landtechnik

Situationsbeschreibung

Für ein System in der landwirtschaftlichen Automatisierung (z.B. eine Fassreinigung, eine Klappvorrichtung oder ein Gebläse) soll eine Ablaufsteuerung implementiert werden. Es werden 8 Stufen oder Aktoren nacheinander geschaltet.

Die Steuerung soll über den Baustein `sequence_ET_08_AX` realisiert werden.

Funktionsbeschreibung der Ablaufsteuerung

- **Start:** Durch Betätigen des Tasters `I1` (`START_S1`).
- **Phasen:** 8 aufeinanderfolgende Ausgangsschritte (Ausgänge `Q1` bis `Q8`).
- **Übergänge:** Die Schrittübergänge sind eine Kombination aus Event und Zeitsteuerung.

- **Reset:** Zu jedem Zeitpunkt kann die Schrittkette über einen Reset (z.B. Taster `I4` oder Softkey) in den Ausgangszustand (Schritt 0) zurückgesetzt werden.

Arbeitsauftrag

1. Öffnen Sie das 4diac-Projekt und legen Sie die SubApp `Uebung_045_AX` bzw. `Uebung_045` an.
2. Platzieren Sie den Sequenzer-Baustein `sequence_ET_08_AX` und konfigurieren Sie die Parameter.
3. Verbinden Sie den Start-Taster `I1` und den Reset-Taster `I4` mit den entsprechenden Eingängen.
4. Schalten Sie die 8 Ausgänge auf die physischen Ausgänge `Output_Q1` bis `Output_Q8`.
5. Zur Visualisierung soll die Zustandsnummer `STATE_NR` auf dem ISOBUS Virtual Terminal-Ausgang `OutputNumber_N1` angezeigt werden.

2.1.2.314 Uebung_046_AX: Ablaufsteuerung 8-Kanal (Event & Zeit) ENDLOS (Adapter Version)

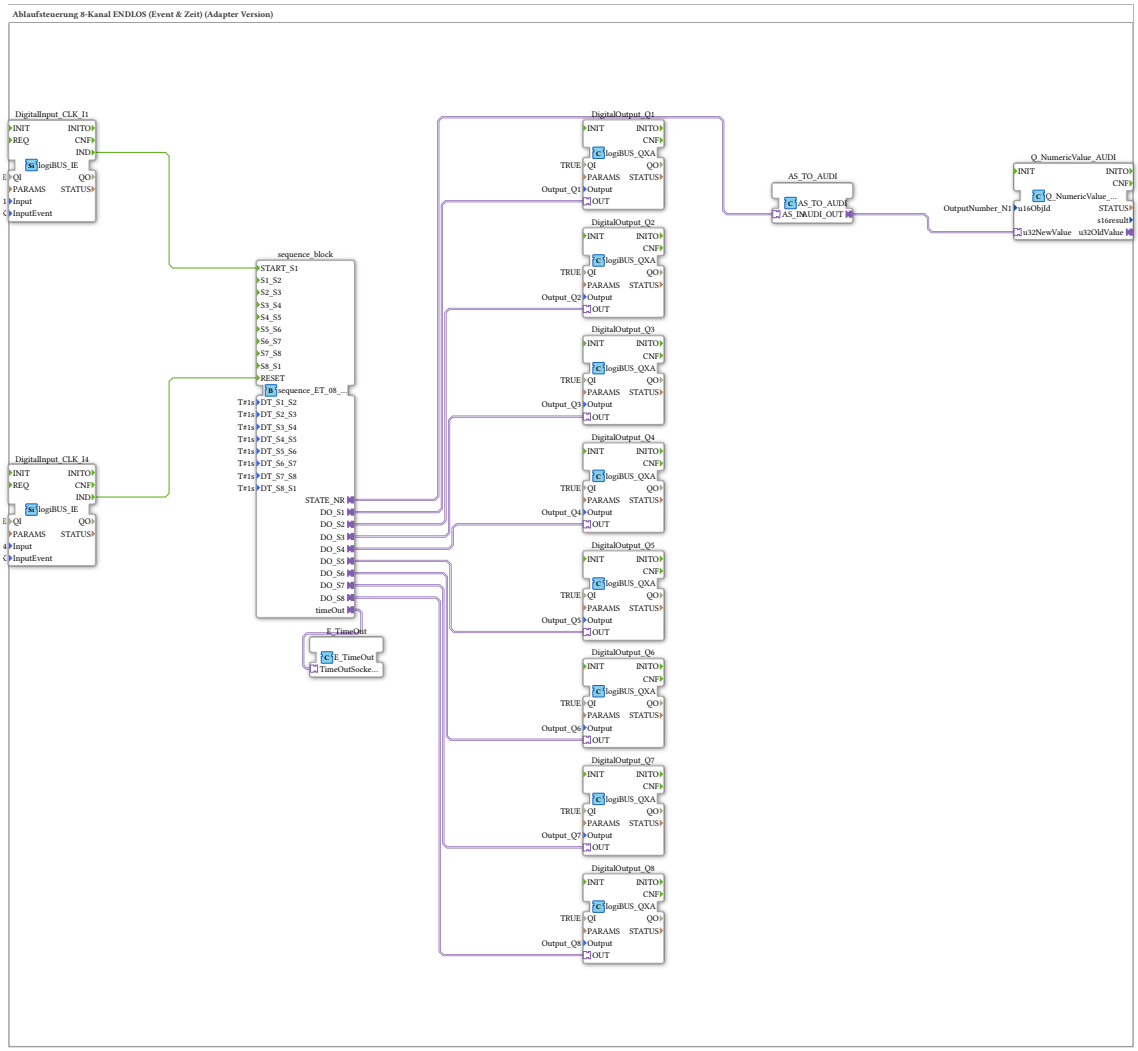
Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_046_AX`. Wir bauen eine kombinierte Event- und Zeit-Schrittkette mit AX-Adaptertechnologie.

Ziel der Übung

Realisierung einer Abfolge von 8 Ausgängen (Q1, Q2, Q3, Q4, Q5, Q6, Q7, Q8), gestartet und zurückgesetzt über Ereignisse (Taster), wobei die einzelnen Schritte zusätzlich über feste Zeiten (`DT_Sx_Sy`, je `T#1s`) weitergeschaltet werden - eine Kombination aus Start/Reset per Taster und automatischem Zeittakt zwischen den Schritten.

Beschreibung und Komponenten

Die Subapplikation `Uebung_046_AX` nutzt den kombinierten Sequenzer `sequence_ET_08_loop_AX`.



- **`sequence_ET_08_loop_AX`**: Startet auf Ereignis (`I1`), schaltet die 8 Schritte danach zeitgesteuert weiter.
- **`Q_NumericValue_AUDI`**: Zeigt die aktuelle Schrittnummer auf dem ISOBUS-Terminal an.

Funktionsweise

1. Start durch Taster `I1` -> Sequenz springt zu `S1`.
2. Jeder Ausgang (`Q1, Q2, Q3, Q4, Q5, Q6, Q7, Q8`) wird für die eingestellte Zeit aktiv geschaltet, bevor automatisch zum nächsten Schritt gewechselt wird.
3. Nach dem letzten Schritt kehrt die Sequenz automatisch zu S1 zurück (ENDLOS).
4. Reset durch Taster `I4` -> Alles aus.

2.1.2.315 Übung 046: Ablaufsteuerung 8-Kanal ENDLOS (Event & Zeit)

Thema: Ablaufsteuerungen in der Landtechnik

Situationsbeschreibung

Für ein System in der landwirtschaftlichen Automatisierung (z.B. eine Fassreinigung, eine Klappvorrichtung oder ein Gebläse) soll eine Ablaufsteuerung implementiert werden. Es werden 8 Stufen oder Aktoren nacheinander geschaltet.

Die Steuerung soll über den Baustein `sequence_ET_08_loop_AX` realisiert werden.

Funktionsbeschreibung der Ablaufsteuerung

- **Start:** Durch Betätigen des Tasters `I1` (`START_S1`).
- **Phasen:** 8 aufeinanderfolgende Ausgangsschritte (Ausgänge `Q1` bis `Q8`).
- **Übergänge:** Die Schrittübergänge sind eine Kombination aus Event und Zeitsteuerung.

- **Reset:** Zu jedem Zeitpunkt kann die Schrittkette über einen Reset (z.B. Taster `I4` oder Softkey) in den Ausgangszustand (Schritt 0) zurückgesetzt werden.

Arbeitsauftrag

1. Öffnen Sie das 4diac-Projekt und legen Sie die SubApp `Uebung_046_AX` bzw. `Uebung_046` an.
2. Platzieren Sie den Sequenzer-Baustein `sequence_ET_08_loop_AX` und konfigurieren Sie die Parameter.
3. Verbinden Sie den Start-Taster `I1` und den Reset-Taster `I4` mit den entsprechenden Eingängen.
4. Schalten Sie die 8 Ausgänge auf die physischen Ausgänge `Output_Q1` bis `Output_Q8`.
5. Zur Visualisierung soll die Zustandsnummer `STATE_NR` auf dem ISOBUS Virtual Terminal-Ausgang `OutputNumber_N1` angezeigt werden.

2.1.2.316 Uebung_047_AX: Reine Event-Schrittkette 4-Kanal (Adapter Version)

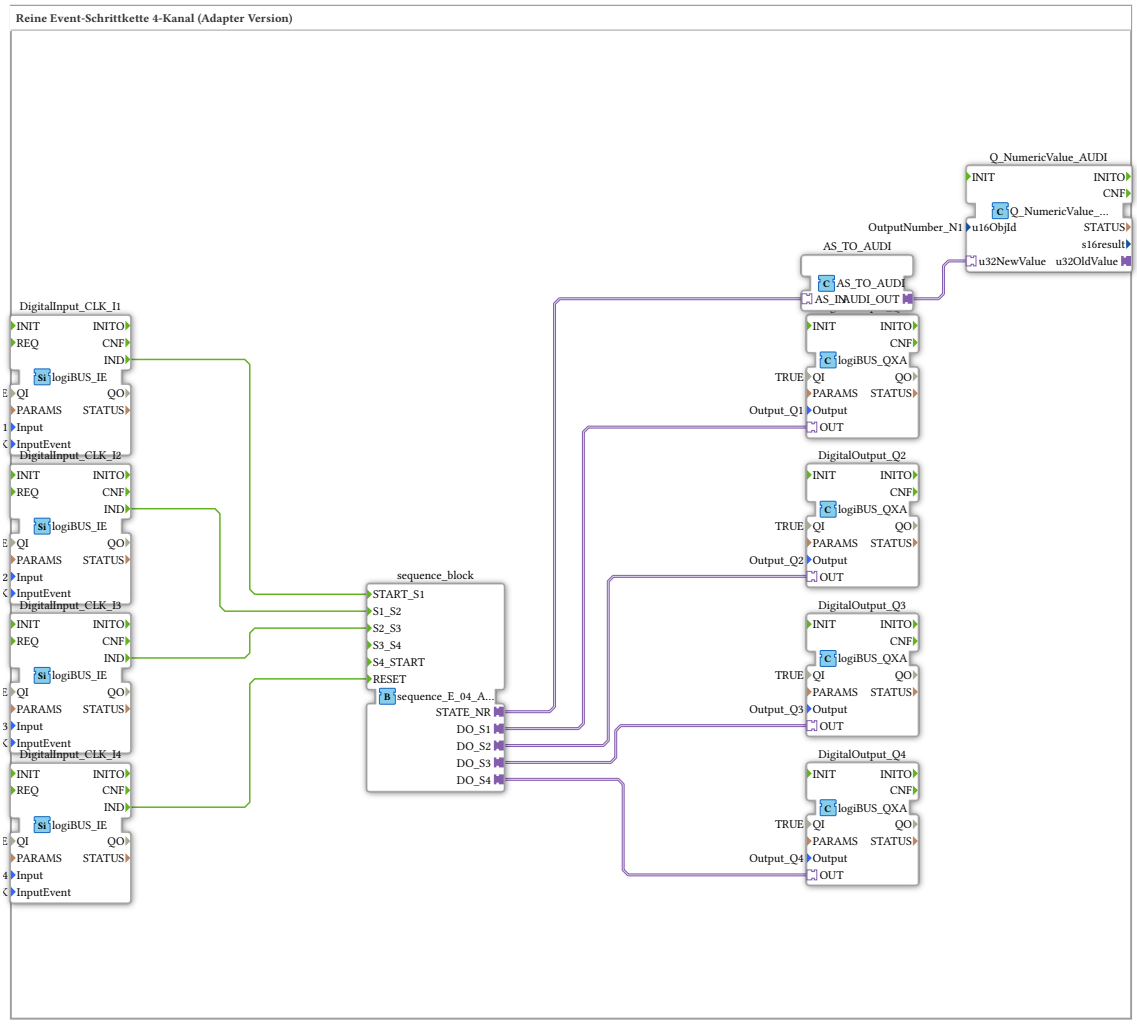
Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_047_AX`. Wir bauen eine rein ereignisgesteuerte Schrittkette mit AX-Adaptertechnologie.

Ziel der Übung

Realisierung einer Abfolge von 4 Ausgängen (Q1, Q2, Q3, Q4), bei der jeder Schritt ausschließlich durch ein externes Ereignis (Tasterdruck) weitergeschaltet wird - keine automatische Zeitsteuerung.

Beschreibung und Komponenten

Die Subapplikation `Uebung_047_AX` nutzt den Sequenzer `sequence_E_04_AX`, der 4 Zustände rein ereignisgesteuert durchläuft.



Jeder Übergang zwischen den Schritten wird durch eine eigene Tasterbetätigung (`I1` bis `I4`) ausgelöst - die Sequenz wartet an jedem Schritt, bis der nächste Taster gedrückt wird.

- **`Q_NumericValue_AUDI`**: Zeigt die aktuelle Schrittnummer auf dem ISOBUS-Terminal an.

Funktionsweise

1. Start durch Taster `I1` -> Sequenz springt zu `S1`, Ausgang `Q1` wird aktiv.
2. Jeder weitere Tasterdruck schaltet zum nächsten Schritt weiter.
3. Nach dem letzten Schritt bleibt die Sequenz stehen, bis sie zurückgesetzt wird.
4. Reset durch den letzten Taster -> Alles aus.

2.1.2.317 Übung 047: Reine Event-Schrittkette 4-Kanal

Thema: Ablaufsteuerungen in der Landtechnik

Situationsbeschreibung

Für ein System in der landwirtschaftlichen Automatisierung (z.B. eine Fassreinigung, eine Klappvorrichtung oder ein Gebläse) soll eine Ablaufsteuerung implementiert werden. Es werden 4 Stufen oder Aktoren nacheinander geschaltet.

Die Steuerung soll über den Baustein `sequence_E_04_AX` realisiert werden.

Funktionsbeschreibung der Ablaufsteuerung

- **Start:** Durch Betätigen des Tasters `I1` (`START_S1`).
- **Phasen:** 4 aufeinanderfolgende Ausgangsschritte (Ausgänge `Q1` bis `Q4`).
- **Übergänge:** Die Schrittübergänge laufen rein ereignisgesteuert ab (Taster I2, I3 etc.).
- **Reset:** Zu jedem Zeitpunkt kann die Schrittkette über einen Reset (z.B. Taster `I4` oder Softkey) in den Ausgangszustand (Schritt 0) zurückgesetzt werden.

Arbeitsauftrag

1. Öffnen Sie das 4diac-Projekt und legen Sie die SubApp `Uebung_047_AX` bzw. `Uebung_047` an.
2. Platzieren Sie den Sequenzer-Baustein `sequence_E_04_AX` und konfigurieren Sie die Parameter.
3. Verbinden Sie den Start-Taster `I1` und den Reset-Taster `I4` mit den entsprechenden Eingängen.
4. Schalten Sie die 4 Ausgänge auf die physischen Ausgänge `Output_Q1` bis `Output_Q4`.
5. Zur Visualisierung soll die Zustandsnummer `STATE_NR` auf dem ISOBUS Virtual Terminal-Ausgang `OutputNumber_N1` angezeigt werden.

2.1.2.318 Uebung_048_AX: Reine Event-Schrittkette 4-Kanal ENDLOS (Adapter Version)

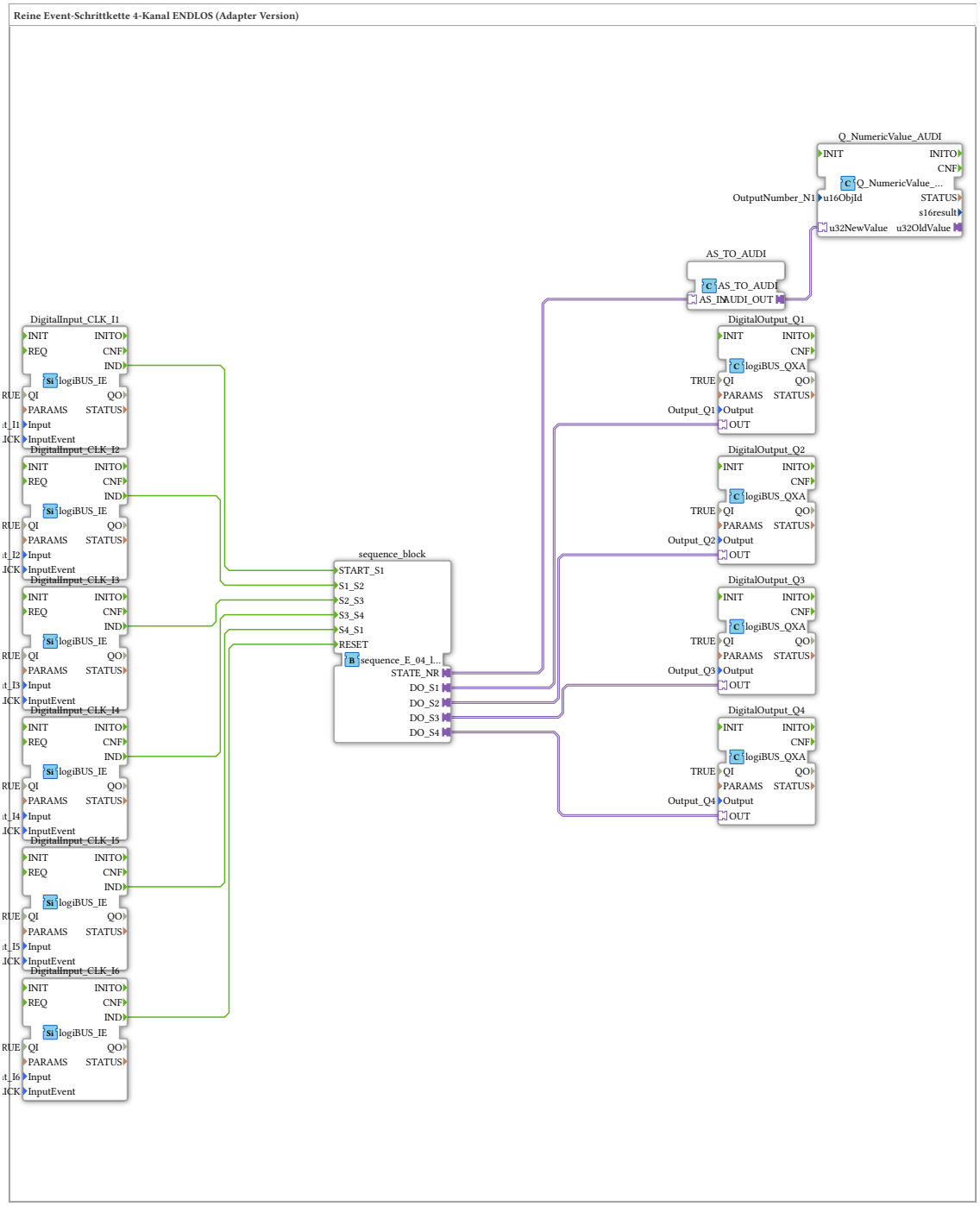
Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_048_AX`. Wir bauen eine rein ereignisgesteuerte Schrittkette mit AX-Adaptertechnologie.

Ziel der Übung

Realisierung einer Abfolge von 4 Ausgängen (Q1, Q2, Q3, Q4), bei der jeder Schritt ausschließlich durch ein externes Ereignis (Tasterdruck) weitergeschaltet wird - keine automatische Zeitsteuerung.

Beschreibung und Komponenten

Die Subapplikation `Uebung_048_AX` nutzt den Sequenzer `sequence_E_04_loop_AX`, der 4 Zustände rein ereignisgesteuert durchläuft.



Jeder Übergang zwischen den Schritten wird durch eine eigene Tasterbetätigung (`I1` bis `I6`) ausgelöst - die Sequenz wartet an jedem Schritt, bis der nächste Taster gedrückt wird.

- **`Q_NumericValue_AUDI`**: Zeigt die aktuelle Schrittnummer auf dem ISOBUS-Terminal an.

Funktionsweise

1. Start durch Taster `I1` -> Sequenz springt zu `S1`, Ausgang `Q1` wird aktiv.
2. Jeder weitere Tasterdruck schaltet zum nächsten Schritt weiter.
3. Nach dem letzten Schritt kehrt die Sequenz automatisch zu S1 zurück (ENDLOS).
4. Reset durch den letzten Taster -> Alles aus.

2.1.2.319 Übung 048: Reine Event-Schrittkette 4-Kanal ENDLOS

Thema: Ablaufsteuerungen in der Landtechnik

Situationsbeschreibung

Für ein System in der landwirtschaftlichen Automatisierung (z.B. eine Fassreinigung, eine Klappvorrichtung oder ein Gebläse) soll eine Ablaufsteuerung implementiert werden. Es werden 4 Stufen oder Aktoren nacheinander geschaltet.

Die Steuerung soll über den Baustein `sequence_E_04_loop_AX` realisiert werden.

Funktionsbeschreibung der Ablaufsteuerung

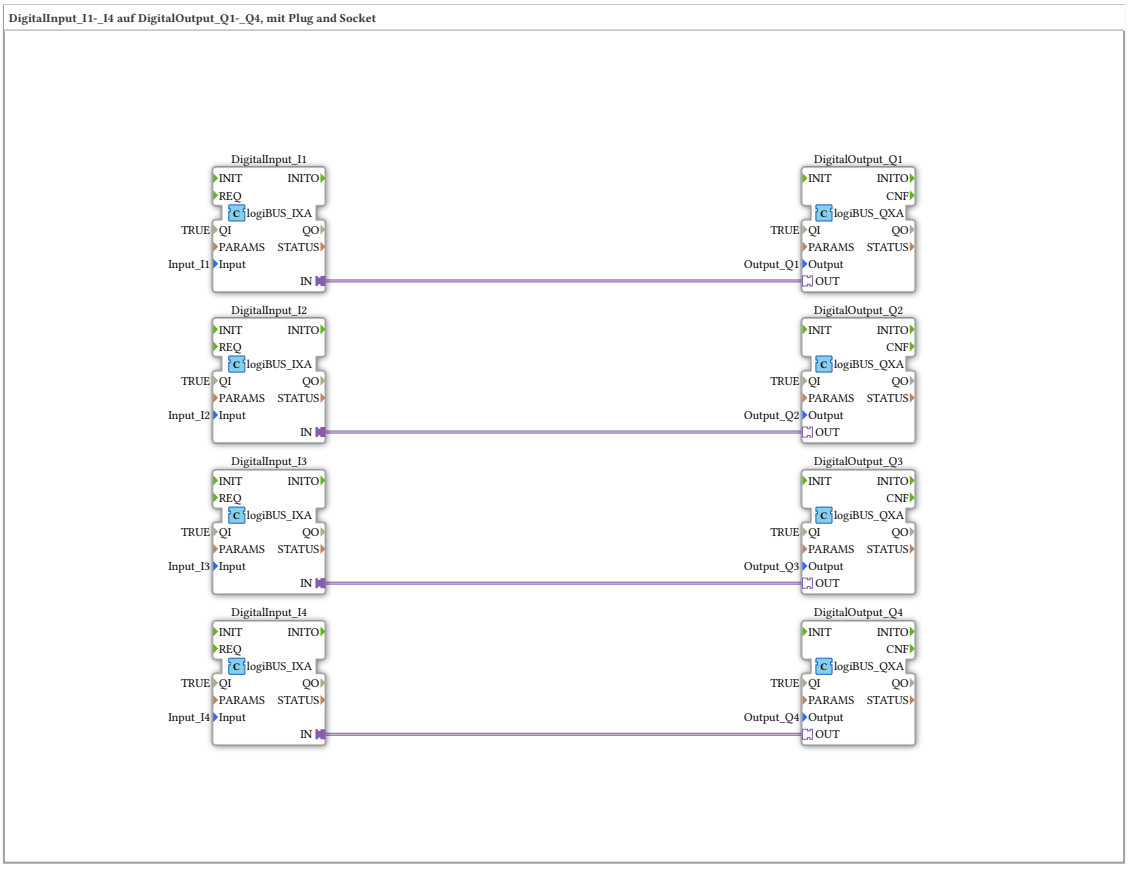
- **Start:** Durch Betätigen des Tasters `I1` (`START_S1`).
- **Phasen:** 4 aufeinanderfolgende Ausgangsschritte (Ausgänge `Q1` bis `Q4`).
- **Übergänge:** Die Schrittübergänge laufen rein ereignisgesteuert ab (Taster I2, I3 etc.).
- **Reset:** Zu jedem Zeitpunkt kann die Schrittkette über einen Reset (z.B. Taster `I4` oder Softkey) in den Ausgangszustand (Schritt 0) zurückgesetzt werden.

Arbeitsauftrag

1. Öffnen Sie das 4diac-Projekt und legen Sie die SubApp `Uebung_048_AX` bzw. `Uebung_048` an.
2. Platzieren Sie den Sequenzer-Baustein `sequence_E_04_loop_AX` und konfigurieren Sie die Parameter.
3. Verbinden Sie den Start-Taster `I1` und den Reset-Taster `I4` mit den entsprechenden Eingängen.
4. Schalten Sie die 4 Ausgänge auf die physischen Ausgänge `Output_Q1` bis `Output_Q4`.
5. Zur Visualisierung soll die Zustandsnummer `STATE_NR` auf dem ISOBUS Virtual Terminal-Ausgang `OutputNumber_N1` angezeigt werden.

2.1.2.320 Uebung_049_AX: DigitalInput_I1-_I4 auf DigitalOutput_Q1-_Q4, mit Plug and Socket

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_049_AX (DigitalInput_I1-_I4 auf DigitalOutput_Q1-_Q4, mit Plug and Socket).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **DigitalInput_I1-_I4 auf DigitalOutput_Q1-_Q4, mit Plug and Socket**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_049_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
- **DigitalInput_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
- **DigitalOutput_Q2**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q2
- **DigitalInput_I2**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I2
- **DigitalOutput_Q3**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q3
- **DigitalInput_I3**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I3
- **DigitalOutput_Q4**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q4
- **DigitalInput_I4**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I4

Verbindungen und Schnittstellen

Adapterverbindungen:

- DigitalInput_I1.IN -> DigitalOutput_Q1.OUT
- DigitalInput_I2.IN -> DigitalOutput_Q2.OUT
- DigitalInput_I3.IN -> DigitalOutput_Q3.OUT
- DigitalInput_I4.IN -> DigitalOutput_Q4.OUT

Programmablauf und Funktionsweise

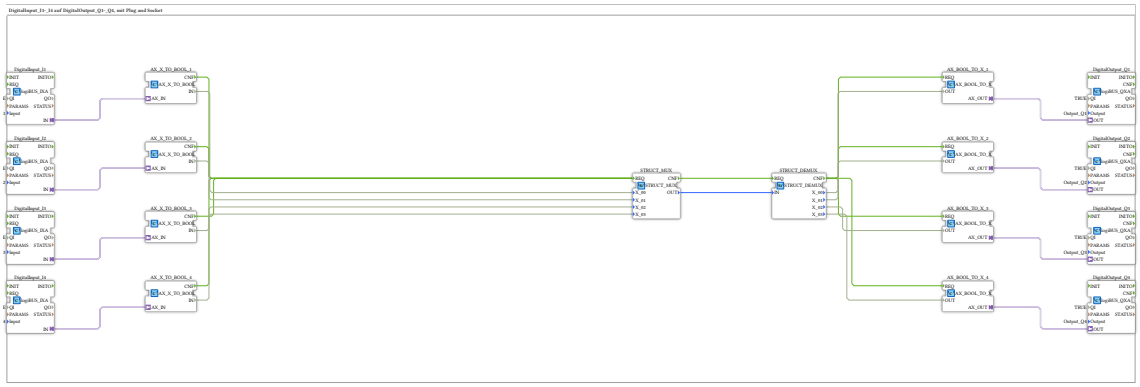
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_049_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.321 Uebung_051_AX: DigitalInput_I1-_I4 auf DigitalOutput_Q1-_Q4, mit Plug and Socket

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_051_AX (DigitalInput_I1-_I4 auf DigitalOutput_Q1-_Q4, mit Plug and Socket).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **DigitalInput_I1-_I4 auf DigitalOutput_Q1-_Q4, mit Plug and Socket**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_051_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
- **DigitalInput_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
- **DigitalOutput_Q2**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q2
- **DigitalInput_I2**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I2
- **DigitalOutput_Q3**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q3
- **DigitalInput_I3**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I3
- **DigitalOutput_Q4**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q4
- **DigitalInput_I4**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I4
- **STRUCT_DEMUX**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **STRUCT_MUX**: Instanz des Typs eclipse4diac::convert::STRUCT_MUX.
- **AX_X_TO_BOOL_1**: Instanz des Typs adapter::conversion::unidirectional::AX_X_TO_BOOL.
- **AX_X_TO_BOOL_2**: Instanz des Typs adapter::conversion::unidirectional::AX_X_TO_BOOL.
- **AX_X_TO_BOOL_3**: Instanz des Typs adapter::conversion::unidirectional::AX_X_TO_BOOL.
- **AX_X_TO_BOOL_4**: Instanz des Typs adapter::conversion::unidirectional::AX_X_TO_BOOL.
- **AX_BOOL_TO_X_1**: Instanz des Typs adapter::conversion::unidirectional::AX_BOOL_TO_X.
- **AX_BOOL_TO_X_2**: Instanz des Typs adapter::conversion::unidirectional::AX_BOOL_TO_X.
- **AX_BOOL_TO_X_3**: Instanz des Typs adapter::conversion::unidirectional::AX_BOOL_TO_X.
- **AX_BOOL_TO_X_4**: Instanz des Typs adapter::conversion::unidirectional::AX_BOOL_TO_X.

Verbindungen und Schnittstellen

Adapterverbindungen:

- DigitalInput_I1.IN -> AX_X_TO_BOOL_1.AX_IN
- DigitalInput_I2.IN -> AX_X_TO_BOOL_2.AX_IN
- DigitalInput_I3.IN -> AX_X_TO_BOOL_3.AX_IN
- DigitalInput_I4.IN -> AX_X_TO_BOOL_4.AX_IN
- AX_BOOL_TO_X_1.AX_OUT -> DigitalOutput_Q1.OUT
- AX_BOOL_TO_X_2.AX_OUT -> DigitalOutput_Q2.OUT
- AX_BOOL_TO_X_3.AX_OUT -> DigitalOutput_Q3.OUT
- AX_BOOL_TO_X_4.AX_OUT -> DigitalOutput_Q4.OUT

Ereignisverbindungen:

- AX_X_TO_BOOL_1.CNF -> STRUCT_MUX.REQ
- AX_X_TO_BOOL_2.CNF -> STRUCT_MUX.REQ
- AX_X_TO_BOOL_3.CNF -> STRUCT_MUX.REQ
- AX_X_TO_BOOL_4.CNF -> STRUCT_MUX.REQ
- STRUCT_MUX.CNF -> STRUCT_DEMUX.REQ
- STRUCT_DEMUX.CNF -> AX_BOOL_TO_X_1.REQ
- STRUCT_DEMUX.CNF -> AX_BOOL_TO_X_2.REQ
- STRUCT_DEMUX.CNF -> AX_BOOL_TO_X_3.REQ
- STRUCT_DEMUX.CNF -> AX_BOOL_TO_X_4.REQ

Datenverbindungen:

- AX_X_TO_BOOL_1.IN -> STRUCT_MUX.X_00
- AX_X_TO_BOOL_2.IN -> STRUCT_MUX.X_01
- AX_X_TO_BOOL_3.IN -> STRUCT_MUX.X_02
- AX_X_TO_BOOL_4.IN -> STRUCT_MUX.X_03
- STRUCT_MUX.OUT -> STRUCT_DEMUX.IN
- STRUCT_DEMUX.X_00 -> AX_BOOL_TO_X_1.OUT
- STRUCT_DEMUX.X_01 -> AX_BOOL_TO_X_2.OUT
- STRUCT_DEMUX.X_02 -> AX_BOOL_TO_X_3.OUT
- STRUCT_DEMUX.X_03 -> AX_BOOL_TO_X_4.OUT

Programmablauf und Funktionsweise

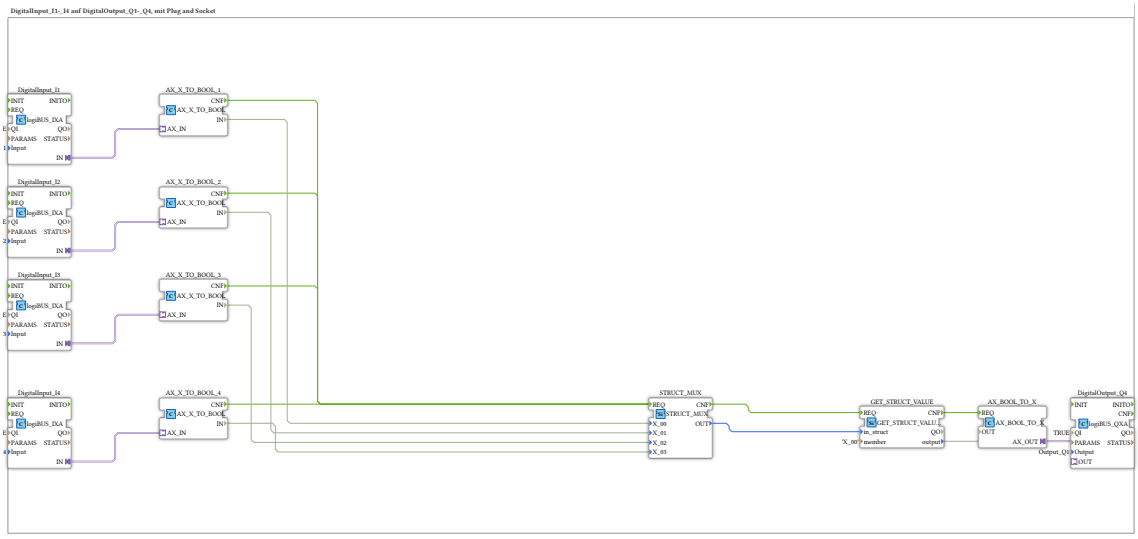
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_051_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.322 Uebung_052_AX: DigitalInput_I1-_I4 auf DigitalOutput_Q1-_Q4, mit Plug and Socket

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_052_AX (DigitalInput_I1-_I4 auf DigitalOutput_Q1-_Q4, mit Plug and Socket).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **DigitalInput_I1-_I4 auf DigitalOutput_Q1-_Q4, mit Plug and Socket**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_052_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
- **DigitalInput_I2**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I2
- **DigitalInput_I3**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I3
- **DigitalOutput_Q4**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
- **DigitalInput_I4**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I4
- **GET_STRUCT_VALUE**: Instanz des Typs eclipse4diac::convert::GET_STRUCT_VALUE.
  - Parameter member = ‚X_00‘
- **STRUCT_MUX**: Instanz des Typs eclipse4diac::convert::STRUCT_MUX.
- **AX_X_TO_BOOL_1**: Instanz des Typs adapter::conversion::unidirectional::AX_X_TO_BOOL.
- **AX_X_TO_BOOL_2**: Instanz des Typs adapter::conversion::unidirectional::AX_X_TO_BOOL.
- **AX_X_TO_BOOL_3**: Instanz des Typs adapter::conversion::unidirectional::AX_X_TO_BOOL.
- **AX_X_TO_BOOL_4**: Instanz des Typs adapter::conversion::unidirectional::AX_X_TO_BOOL.
- **AX_BOOL_TO_X**: Instanz des Typs adapter::conversion::unidirectional::AX_BOOL_TO_X.

Verbindungen und Schnittstellen

Adapterverbindungen:

- DigitalInput_I1.IN -> AX_X_TO_BOOL_1.AX_IN
- DigitalInput_I2.IN -> AX_X_TO_BOOL_2.AX_IN
- DigitalInput_I3.IN -> AX_X_TO_BOOL_3.AX_IN
- DigitalInput_I4.IN -> AX_X_TO_BOOL_4.AX_IN

- AX_BOOL_TO_X.AX_OUT -> DigitalOutput_Q4.OUT

Ereignisverbindungen:

- GET_STRUCT_VALUE.CNF -> AX_BOOL_TO_X.REQ
- AX_X_TO_BOOL_1.CNF -> STRUCT_MUX.REQ
- AX_X_TO_BOOL_2.CNF -> STRUCT_MUX.REQ
- AX_X_TO_BOOL_3.CNF -> STRUCT_MUX.REQ
- AX_X_TO_BOOL_4.CNF -> STRUCT_MUX.REQ
- STRUCT_MUX.CNF -> GET_STRUCT_VALUE.REQ

Datenverbindungen:

- GET_STRUCT_VALUE.output -> AX_BOOL_TO_X.OUT
- AX_X_TO_BOOL_1.IN -> STRUCT_MUX.X_00
- AX_X_TO_BOOL_2.IN -> STRUCT_MUX.X_01
- AX_X_TO_BOOL_3.IN -> STRUCT_MUX.X_02
- AX_X_TO_BOOL_4.IN -> STRUCT_MUX.X_03
- STRUCT_MUX.OUT -> GET_STRUCT_VALUE.in_struct

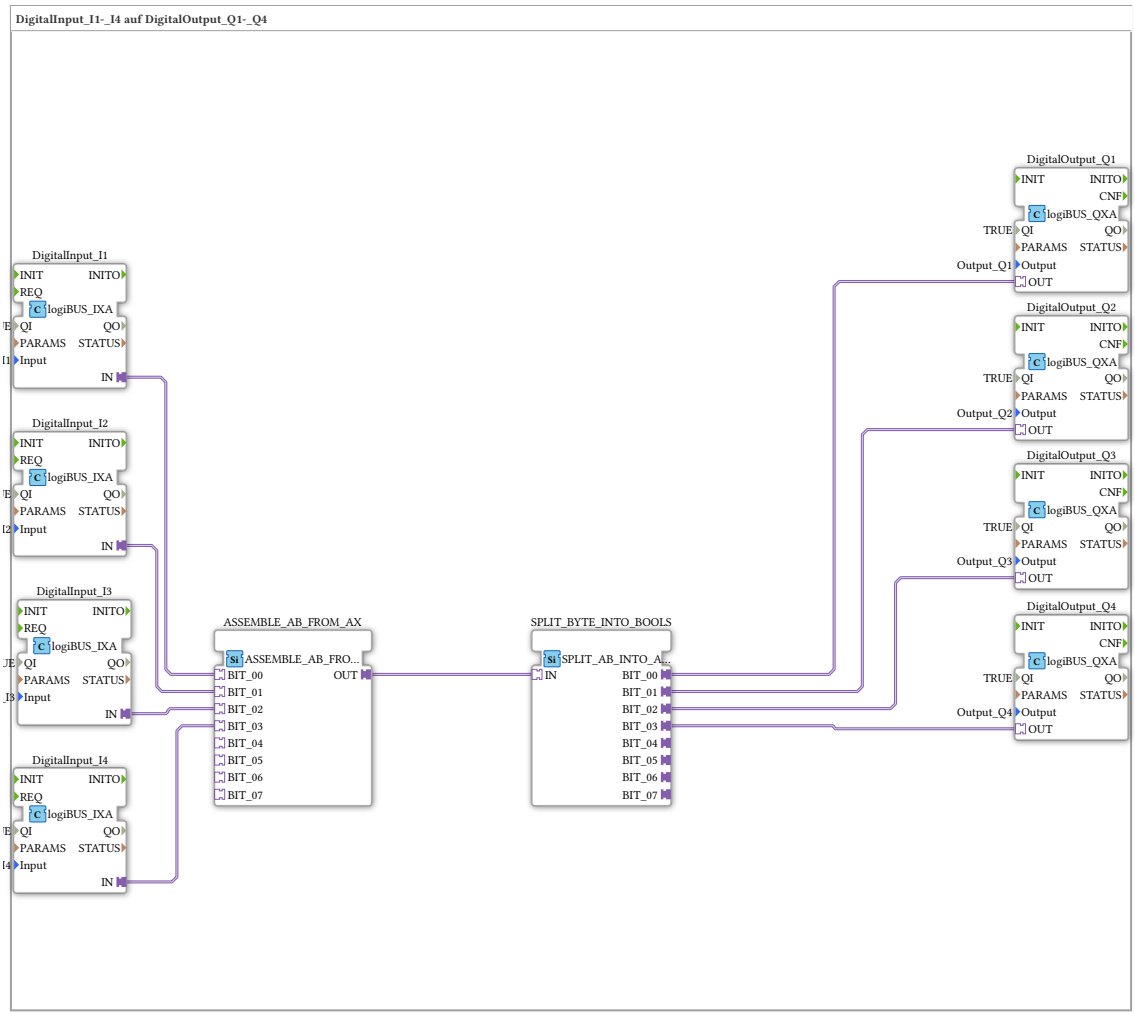
Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_052_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.323 Uebung_053_AB: DigitalInput_I1-_I4 auf DigitalOutput_Q1-_Q4



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung zeigt, wie vier digitale Eingänge (I1 bis I4) über einen Adapter zu einem Byte zusammengefasst und anschließend wieder auf vier digitale Ausgänge (Q1 bis Q4) aufgeteilt werden. Es wird demonstriert, wie mit Hilfe der Bausteine `ASSEMBLE_AB_FROM_AX` und `SPLIT_AB_INTO_AX` parallele Binärsignale in einen Datenbus umgewandelt und zurückgewonnen werden können.

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_I1 .. DigitalInput_I4** (Typ `logiBUS::io::DI::logiBUS_IXA`) Lesen der physikalischen Eingänge **Input_I1** bis **Input_I4**. Jeder dieser Bausteine ist mit `QI=TRUE` aktiviert.
- **DigitalOutput_Q1 .. DigitalOutput_Q4** (Typ `logiBUS::io::DQ::logiBUS_QXA`) Schreiben der physikalischen Ausgänge **Output_Q1** bis **Output_Q4**. Auch hier ist `QI=TRUE` gesetzt.
- **ASSEMBLE_BYTE_FROM_BOOLS** (Typ `adapter::assembling::ASSEMBLE_AB_FROM_AX`) Setzt aus den vier booleschen Werten an den Adapter-Sockets `BIT_00` bis `BIT_03` ein Byte zusammen und gibt es am Socket `OUT` aus.
  - **Parameter**: Keine
  - **Eingänge (Adapter)**: `BIT_00`, `BIT_01`, `BIT_02`, `BIT_03` (je ein Boolean)
  - **Ausgang (Adapter)**: `OUT` (Byte)
- **SPLIT_BYTE_INTO_BOOLS** (Typ `adapter::splitting::SPLIT_AB_INTO_AX`) Nimmt ein Byte am Adapter-Socket `IN` entgegen und trennt die Bits einzeln an den Sockets `BIT_00` bis `BIT_03` wieder auf.
  - **Parameter**: Keine

- **Eingang (Adapter)**: `IN` (Byte)
- **Ausgänge (Adapter)**: `BIT_00`, `BIT_01`, `BIT_02`, `BIT_03` (je ein Boolean)

Sub-Bausteine: (keine)

Die Übung verwendet ausschließlich die oben genannten Standard-FBs. Es sind keine weiteren Sub-Applikationen eingebettet.

Programmablauf und Verbindungen

1. Die **digitalen Eingänge** I1 bis I4 werden über die Bausteine `DigitalInput_I1` bis `DigitalInput_I4` eingelesen. Ihre Ausgänge (`IN`) liefern die booleschen Zustände der angeschlossenen Sensoren.
2. Diese vier Bool-Werte werden über Adapter-Verbindungen zu den Sockets `BIT_00` bis `BIT_03` des Bausteins **`ASSEMBLE_BYTE_FROM_BOOLS`** geleitet.
   - `DigitalInput_I1.IN` ⟶ `ASSEMBLE_BYTE_FROM_BOOLS.BIT_00`
   - `DigitalInput_I2.IN` ⟶ `ASSEMBLE_BYTE_FROM_BOOLS.BIT_01`
   - `DigitalInput_I3.IN` ⟶ `ASSEMBLE_BYTE_FROM_BOOLS.BIT_02`
   - `DigitalInput_I4.IN` ⟶ `ASSEMBLE_BYTE_FROM_BOOLS.BIT_03`
3. Der **Assemble-Baustein** packt die vier Bits in ein Byte (niederwertigstes Bit = BIT_00) und gibt dieses Byte an seinem Ausgang `OUT` aus.
4. Dieser Ausgang ist direkt mit dem Eingang `IN` des Bausteins **`SPLIT_BYTE_INTO_BOOLS`** verbunden.
   - `ASSEMBLE_BYTE_FROM_BOOLS.OUT` ⟶ `SPLIT_BYTE_INTO_BOOLS.IN`
5. Der **Split-Baustein** zerlegt das Byte wieder in vier einzelne boolesche Werte an seinen Ausgangs-Sockets `BIT_00` bis `BIT_03`.
6. Diese Werte werden schließlich auf die **digitalen Ausgänge** Q1 bis Q4 geschaltet:
   - `SPLIT_BYTE_INTO_BOOLS.BIT_00` ⟶ `DigitalOutput_Q1.OUT`
   - `SPLIT_BYTE_INTO_BOOLS.BIT_01` ⟶ `DigitalOutput_Q2.OUT`
   - `SPLIT_BYTE_INTO_BOOLS.BIT_02` ⟶ `DigitalOutput_Q3.OUT`
   - `SPLIT_BYTE_INTO_BOOLS.BIT_03` ⟶ `DigitalOutput_Q4.OUT`

Damit werden die Zustände der Eingänge direkt auf die gleichnamigen Ausgänge übertragen. Die Verwendung der Adapter dient dem Verständnis von Byte-Zusammenführung und -Aufteilung in der 4diac-IDE.

Zusammenfassung

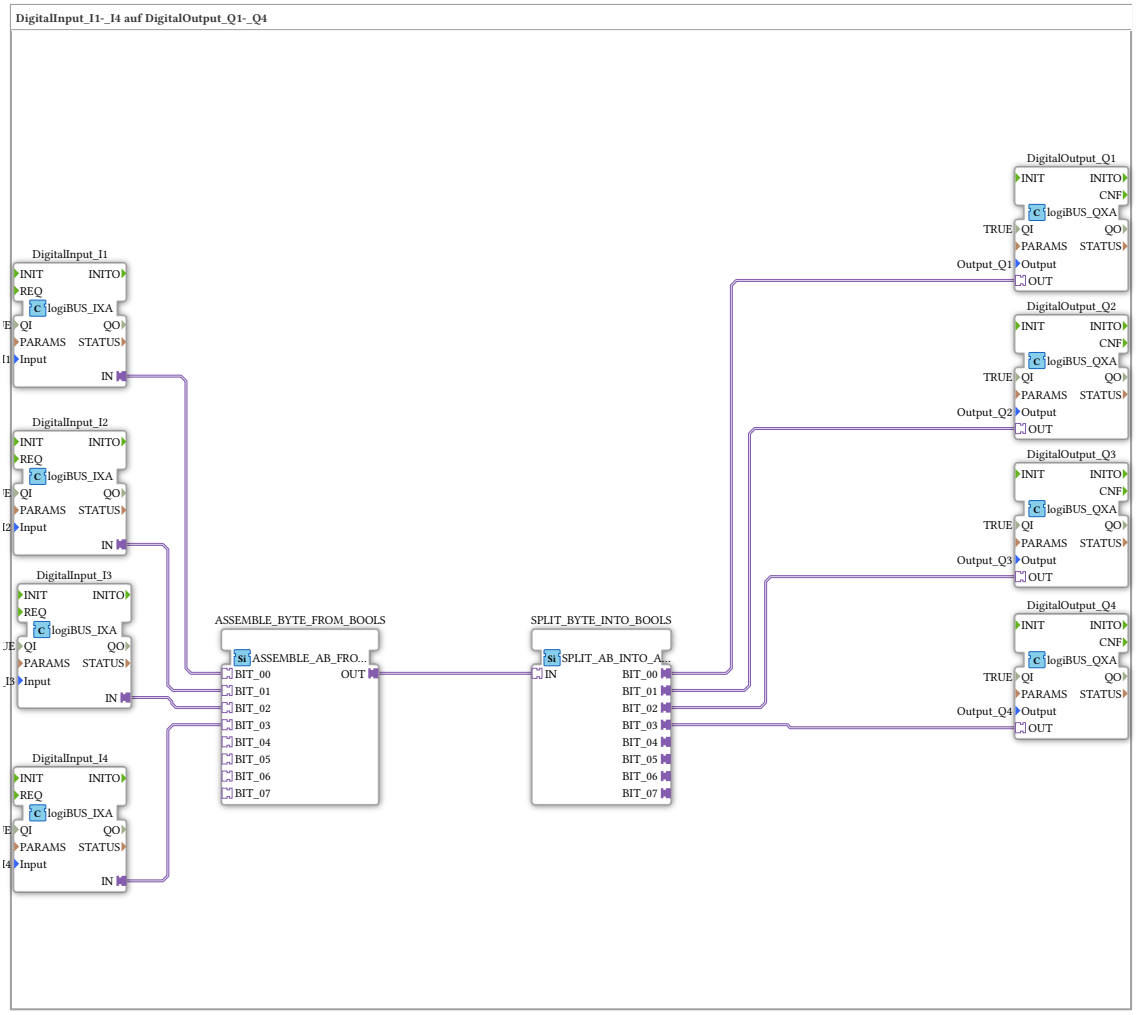
In dieser Übung wird der Datenfluss von vier digitalen Eingängen zu vier digitalen Ausgängen über eine Byte-Adapternetzwerk realisiert. Es werden die Bausteine `ASSEMBLE_AB_FROM_AX` und `SPLIT_AB_INTO_AX` verwendet, um mehrere Bool-Signale zu einem Byte zusammenzufassen und wieder aufzutrennen. Die Übung vermittelt grundlegende Kenntnisse im Umgang mit Adapter-Bausteinen und Datenbus-Strukturen in der IEC 61499-Programmierung.

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

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

2.1.2.324 Uebung_053_AX: DigitalInput_I1-_I4 auf DigitalOutput_Q1-_Q4

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_053_AX (DigitalInput_I1-_I4 auf DigitalOutput_Q1-_Q4).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **DigitalInput_I1-_I4 auf DigitalOutput_Q1-_Q4**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_053_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
- **DigitalInput_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
- **DigitalOutput_Q2**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q2
- **DigitalInput_I2**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I2
- **DigitalOutput_Q3**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q3

- **DigitalInput_I3**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I3
- **DigitalOutput_Q4**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q4
- **DigitalInput_I4**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I4
- **ASSEMBLE_BYTE_FROM_BOOLS**: Instanz des Typs adapter::assembling::ASSEMBLE_AB_FROM_AX.
- **SPLIT_BYTE_INTO_BOOLS**: Instanz des Typs adapter::splitting::SPLIT_AB_INTO_AX.

Verbindungen und Schnittstellen

Adapterverbindungen:

- ASSEMBLE_BYTE_FROM_BOOLS.OUT -> SPLIT_BYTE_INTO_BOOLS.IN
- SPLIT_BYTE_INTO_BOOLS.BIT_00 -> DigitalOutput_Q1.OUT
- SPLIT_BYTE_INTO_BOOLS.BIT_01 -> DigitalOutput_Q2.OUT
- SPLIT_BYTE_INTO_BOOLS.BIT_02 -> DigitalOutput_Q3.OUT
- SPLIT_BYTE_INTO_BOOLS.BIT_03 -> DigitalOutput_Q4.OUT
- DigitalInput_I4.IN -> ASSEMBLE_BYTE_FROM_BOOLS.BIT_03
- DigitalInput_I3.IN -> ASSEMBLE_BYTE_FROM_BOOLS.BIT_02
- DigitalInput_I2.IN -> ASSEMBLE_BYTE_FROM_BOOLS.BIT_01
- DigitalInput_I1.IN -> ASSEMBLE_BYTE_FROM_BOOLS.BIT_00

Programmablauf und Funktionsweise

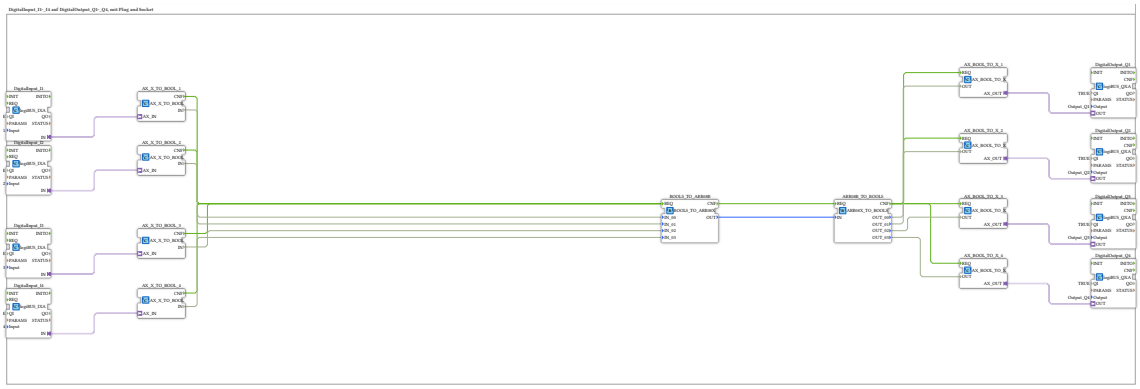
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_053_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.325 Uebung_054_AX: DigitalInput_I1-_I4 auf DigitalOutput_Q1-_Q4, mit Plug and Socket

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_054_AX (DigitalInput_I1-_I4 auf DigitalOutput_Q1-_Q4, mit Plug and Socket).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **DigitalInput_I1-_I4 auf DigitalOutput_Q1-_Q4, mit Plug and Socket**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_054_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
- **DigitalInput_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
- **DigitalOutput_Q2**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q2
- **DigitalInput_I2**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I2
- **DigitalOutput_Q3**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q3
- **DigitalInput_I3**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I3
- **DigitalOutput_Q4**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q4
- **DigitalInput_I4**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I4
- **ARR08B_TO_BOOLS**: Instanz des Typs logiBUS::utils::conversion::arr::forwarding::ARR08X_TO_BOOLS.
- **BOOLS_TO_ARR08B**: Instanz des Typs logiBUS::utils::conversion::arr::reversing::BOOLS_TO_ARR08X.
- **AX_X_TO_BOOL_1**: Instanz des Typs adapter::conversion::unidirectional::AX_X_TO_BOOL.
- **AX_X_TO_BOOL_2**: Instanz des Typs adapter::conversion::unidirectional::AX_X_TO_BOOL.
- **AX_X_TO_BOOL_3**: Instanz des Typs adapter::conversion::unidirectional::AX_X_TO_BOOL.
- **AX_X_TO_BOOL_4**: Instanz des Typs adapter::conversion::unidirectional::AX_X_TO_BOOL.
- **AX_BOOL_TO_X_1**: Instanz des Typs adapter::conversion::unidirectional::AX_BOOL_TO_X.
- **AX_BOOL_TO_X_2**: Instanz des Typs adapter::conversion::unidirectional::AX_BOOL_TO_X.
- **AX_BOOL_TO_X_3**: Instanz des Typs adapter::conversion::unidirectional::AX_BOOL_TO_X.
- **AX_BOOL_TO_X_4**: Instanz des Typs adapter::conversion::unidirectional::AX_BOOL_TO_X.

Verbindungen und Schnittstellen

Adapterverbindungen:

- DigitalInput_I1.IN -> AX_X_TO_BOOL_1.AX_IN
- DigitalInput_I2.IN -> AX_X_TO_BOOL_2.AX_IN
- DigitalInput_I3.IN -> AX_X_TO_BOOL_3.AX_IN
- DigitalInput_I4.IN -> AX_X_TO_BOOL_4.AX_IN
- AX_BOOL_TO_X_1.AX_OUT -> DigitalOutput_Q1.OUT
- AX_BOOL_TO_X_2.AX_OUT -> DigitalOutput_Q2.OUT
- AX_BOOL_TO_X_3.AX_OUT -> DigitalOutput_Q3.OUT
- AX_BOOL_TO_X_4.AX_OUT -> DigitalOutput_Q4.OUT

Ereignisverbindungen:

- BOOLS_TO_ARR08B.CNF -> ARR08B_TO_BOOLS.REQ
- AX_X_TO_BOOL_1.CNF -> BOOLS_TO_ARR08B.REQ
- AX_X_TO_BOOL_2.CNF -> BOOLS_TO_ARR08B.REQ
- AX_X_TO_BOOL_3.CNF -> BOOLS_TO_ARR08B.REQ
- AX_X_TO_BOOL_4.CNF -> BOOLS_TO_ARR08B.REQ

- ARR08B_TO_BOOLS.CNF -> AX_BOOL_TO_X_1.REQ
- ARR08B_TO_BOOLS.CNF -> AX_BOOL_TO_X_2.REQ
- ARR08B_TO_BOOLS.CNF -> AX_BOOL_TO_X_3.REQ
- ARR08B_TO_BOOLS.CNF -> AX_BOOL_TO_X_4.REQ

Datenverbindungen:

- BOOLS_TO_ARR08B.OUT -> ARR08B_TO_BOOLS.IN
- AX_X_TO_BOOL_1.IN -> BOOLS_TO_ARR08B.IN_00
- AX_X_TO_BOOL_2.IN -> BOOLS_TO_ARR08B.IN_01
- AX_X_TO_BOOL_3.IN -> BOOLS_TO_ARR08B.IN_02
- AX_X_TO_BOOL_4.IN -> BOOLS_TO_ARR08B.IN_03
- ARR08B_TO_BOOLS.OUT_00 -> AX_BOOL_TO_X_1.OUT
- ARR08B_TO_BOOLS.OUT_01 -> AX_BOOL_TO_X_2.OUT
- ARR08B_TO_BOOLS.OUT_02 -> AX_BOOL_TO_X_3.OUT
- ARR08B_TO_BOOLS.OUT_03 -> AX_BOOL_TO_X_4.OUT

Programmablauf und Funktionsweise

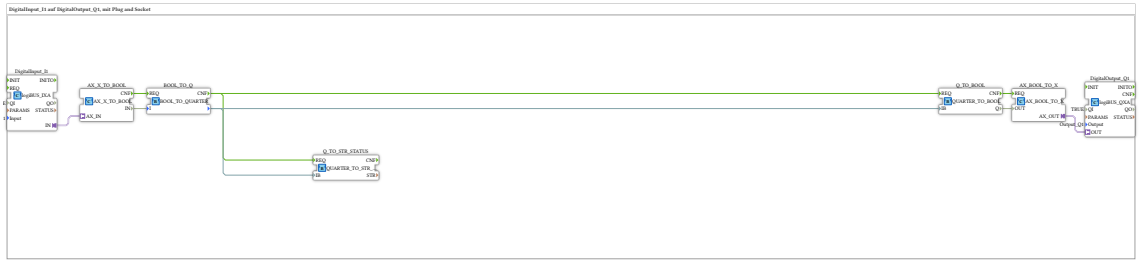
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_054_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.326 Uebung_055_AX: DigitalInput_I1 auf DigitalOutput_Q1, mit Plug and Socket

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_055_AX (DigitalInput_I1 auf DigitalOutput_Q1, mit Plug and Socket).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **DigitalInput_I1 auf DigitalOutput_Q1, mit Plug and Socket**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_055_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
- **DigitalInput_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
- **Q_TO_STR_STATUS**: Instanz des Typs logiBUS::utils::quarter::QUARTER_TO_STR_STATUS.
- **Q_TO_BOOL**: Instanz des Typs logiBUS::utils::quarter::QUARTER_TO_BOOL.

- **BOOL_TO_Q**: Instanz des Typs logiBUS::utils::quarter::BOOL_TO_QUARTER.
- **AX_X_TO_BOOL**: Instanz des Typs adapter::conversion::unidirectional::AX_X_TO_BOOL.
- **AX_BOOL_TO_X**: Instanz des Typs adapter::conversion::unidirectional::AX_BOOL_TO_X.

Verbindungen und Schnittstellen

Adapterverbindungen:

- DigitalInput_I1.IN -> AX_X_TO_BOOL.AX_IN
- AX_BOOL_TO_X.AX_OUT -> DigitalOutput_Q1.OUT

Ereignisverbindungen:

- Q_TO_BOOL.CNF -> AX_BOOL_TO_X.REQ
- AX_X_TO_BOOL.CNF -> BOOL_TO_Q.REQ
- BOOL_TO_Q.CNF -> Q_TO_STR_STATUS.REQ
- BOOL_TO_Q.CNF -> Q_TO_BOOL.REQ

Datenverbindungen:

- Q_TO_BOOL.Q -> AX_BOOL_TO_X.OUT
- AX_X_TO_BOOL.IN -> BOOL_TO_Q.I
- BOOL_TO_Q. -> Q_TO_STR_STATUS.IB
- BOOL_TO_Q. -> Q_TO_BOOL.IB

Programmablauf und Funktionsweise

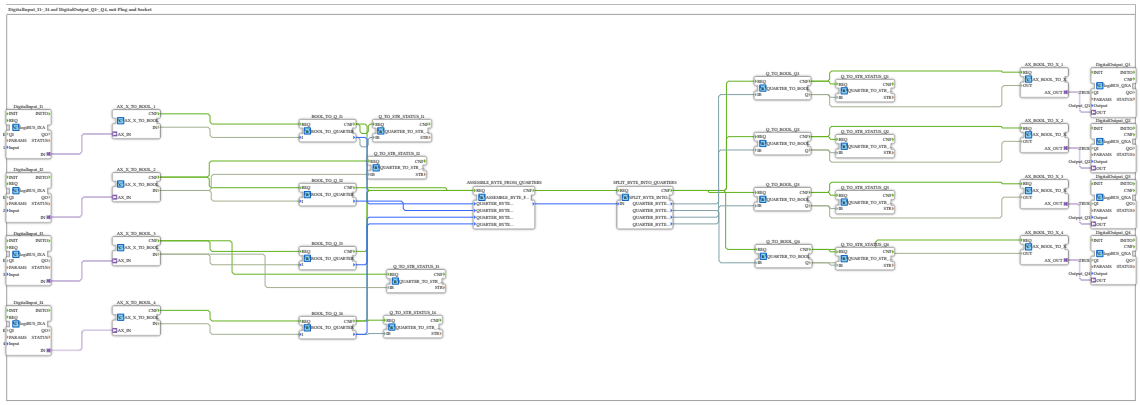
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_055_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.327 Uebung_056_AX: DigitalInput_I1-_I4 auf DigitalOutput_Q1-_Q4, mit Plug and Socket

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_056_AX (DigitalInput_I1-_I4 auf DigitalOutput_Q1-_Q4, mit Plug and Socket).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **DigitalInput_I1-_I4 auf DigitalOutput_Q1-_Q4, mit Plug and Socket**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_056_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
- **ASSEMBLE_BYTE_FROM_QUARTERS**: Instanz des Typs eclipse4diac::utils::assembling::ASSEMBLE_BYTE_FROM_QUARTERS.
- **DigitalInput_I2**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I2
- **DigitalInput_I3**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I3
- **DigitalInput_I4**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I4
- **DigitalOutput_Q2**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q2
- **DigitalOutput_Q3**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q3
- **SPLIT_BYTE_INTO_QUARTERS**: Instanz des Typs eclipse4diac::utils::splitting::SPLIT_BYTE_INTO_QUARTERS.
- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
- **DigitalOutput_Q4**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q4
- **Q_TO_BOOL_Q1**: Instanz des Typs logiBUS::utils::quarter::QUARTER_TO_BOOL.
- **Q_TO_BOOL_Q4**: Instanz des Typs logiBUS::utils::quarter::QUARTER_TO_BOOL.
- **Q_TO_BOOL_Q2**: Instanz des Typs logiBUS::utils::quarter::QUARTER_TO_BOOL.
- **Q_TO_BOOL_Q3**: Instanz des Typs logiBUS::utils::quarter::QUARTER_TO_BOOL.
- **Q_TO_STR_STATUS_Q3**: Instanz des Typs logiBUS::utils::quarter::QUARTER_TO_STR_STATUS.
- **Q_TO_STR_STATUS_Q1**: Instanz des Typs logiBUS::utils::quarter::QUARTER_TO_STR_STATUS.
- **Q_TO_STR_STATUS_I3**: Instanz des Typs logiBUS::utils::quarter::QUARTER_TO_STR_STATUS.
- **Q_TO_STR_STATUS_I1**: Instanz des Typs logiBUS::utils::quarter::QUARTER_TO_STR_STATUS.
- **Q_TO_STR_STATUS_I4**: Instanz des Typs logiBUS::utils::quarter::QUARTER_TO_STR_STATUS.
- **Q_TO_STR_STATUS_Q4**: Instanz des Typs logiBUS::utils::quarter::QUARTER_TO_STR_STATUS.
- **Q_TO_STR_STATUS_Q2**: Instanz des Typs logiBUS::utils::quarter::QUARTER_TO_STR_STATUS.
- **Q_TO_STR_STATUS_I2**: Instanz des Typs logiBUS::utils::quarter::QUARTER_TO_STR_STATUS.
- **BOOL_TO_Q_I1**: Instanz des Typs logiBUS::utils::quarter::BOOL_TO_QUARTER.
- **BOOL_TO_Q_I2**: Instanz des Typs logiBUS::utils::quarter::BOOL_TO_QUARTER.
- **BOOL_TO_Q_I4**: Instanz des Typs logiBUS::utils::quarter::BOOL_TO_QUARTER.
- **BOOL_TO_Q_I3**: Instanz des Typs logiBUS::utils::quarter::BOOL_TO_QUARTER.
- **AX_X_TO_BOOL_1**: Instanz des Typs adapter::conversion::unidirectional::AX_X_TO_BOOL.
- **AX_X_TO_BOOL_2**: Instanz des Typs adapter::conversion::unidirectional::AX_X_TO_BOOL.
- **AX_X_TO_BOOL_3**: Instanz des Typs adapter::conversion::unidirectional::AX_X_TO_BOOL.
- **AX_X_TO_BOOL_4**: Instanz des Typs adapter::conversion::unidirectional::AX_X_TO_BOOL.
- **AX_BOOL_TO_X_1**: Instanz des Typs adapter::conversion::unidirectional::AX_BOOL_TO_X.
- **AX_BOOL_TO_X_2**: Instanz des Typs adapter::conversion::unidirectional::AX_BOOL_TO_X.
- **AX_BOOL_TO_X_3**: Instanz des Typs adapter::conversion::unidirectional::AX_BOOL_TO_X.
- **AX_BOOL_TO_X_4**: Instanz des Typs adapter::conversion::unidirectional::AX_BOOL_TO_X.

Verbindungen und Schnittstellen

Adapterverbindungen:

- DigitalInput_I1.IN -> AX_X_TO_BOOL_1.AX_IN
- DigitalInput_I2.IN -> AX_X_TO_BOOL_2.AX_IN
- DigitalInput_I3.IN -> AX_X_TO_BOOL_3.AX_IN
- DigitalInput_I4.IN -> AX_X_TO_BOOL_4.AX_IN
- AX_BOOL_TO_X_1.AX_OUT -> DigitalOutput_Q1.OUT
- AX_BOOL_TO_X_2.AX_OUT -> DigitalOutput_Q2.OUT
- AX_BOOL_TO_X_3.AX_OUT -> DigitalOutput_Q3.OUT
- AX_BOOL_TO_X_4.AX_OUT -> DigitalOutput_Q4.OUT

Ereignisverbindungen:

- ASSEMBLE_BYTE_FROM_QUARTERS.CNF -> SPLIT_BYTE_INTO_QUARTERS.REQ
- SPLIT_BYTE_INTO_QUARTERS.CNF -> Q_TO_BOOL_Q1.REQ
- Q_TO_BOOL_Q1.CNF -> AX_BOOL_TO_X_1.REQ
- SPLIT_BYTE_INTO_QUARTERS.CNF -> Q_TO_BOOL_Q4.REQ
- Q_TO_BOOL_Q4.CNF -> AX_BOOL_TO_X_4.REQ
- SPLIT_BYTE_INTO_QUARTERS.CNF -> Q_TO_BOOL_Q2.REQ
- Q_TO_BOOL_Q2.CNF -> AX_BOOL_TO_X_2.REQ
- SPLIT_BYTE_INTO_QUARTERS.CNF -> Q_TO_BOOL_Q3.REQ
- Q_TO_BOOL_Q3.CNF -> AX_BOOL_TO_X_3.REQ
- Q_TO_BOOL_Q3.CNF -> Q_TO_STR_STATUS_Q3.REQ
- Q_TO_BOOL_Q1.CNF -> Q_TO_STR_STATUS_Q1.REQ
- AX_X_TO_BOOL_3.CNF -> Q_TO_STR_STATUS_I3.REQ
- Q_TO_BOOL_Q4.CNF -> Q_TO_STR_STATUS_Q4.REQ
- Q_TO_BOOL_Q2.CNF -> Q_TO_STR_STATUS_Q2.REQ
- AX_X_TO_BOOL_2.CNF -> Q_TO_STR_STATUS_I2.REQ
- AX_X_TO_BOOL_1.CNF -> BOOL_TO_Q_I1.REQ
- BOOL_TO_Q_I1.CNF -> ASSEMBLE_BYTE_FROM_QUARTERS.REQ
- BOOL_TO_Q_I1.CNF -> Q_TO_STR_STATUS_I1.REQ
- AX_X_TO_BOOL_2.CNF -> BOOL_TO_Q_I2.REQ
- BOOL_TO_Q_I2.CNF -> ASSEMBLE_BYTE_FROM_QUARTERS.REQ
- AX_X_TO_BOOL_4.CNF -> BOOL_TO_Q_I4.REQ
- BOOL_TO_Q_I4.CNF -> ASSEMBLE_BYTE_FROM_QUARTERS.REQ
- BOOL_TO_Q_I4.CNF -> Q_TO_STR_STATUS_I4.REQ
- AX_X_TO_BOOL_3.CNF -> BOOL_TO_Q_I3.REQ
- BOOL_TO_Q_I3.CNF -> ASSEMBLE_BYTE_FROM_QUARTERS.REQ

Datenverbindungen:

- Q_TO_BOOL_Q1.Q -> AX_BOOL_TO_X_1.OUT
- Q_TO_BOOL_Q4.Q -> AX_BOOL_TO_X_4.OUT
- Q_TO_BOOL_Q2.Q -> AX_BOOL_TO_X_2.OUT
- Q_TO_BOOL_Q3.Q -> AX_BOOL_TO_X_3.OUT
- Q_TO_BOOL_Q3.Q -> Q_TO_STR_STATUS_Q3.IB
- Q_TO_BOOL_Q1.Q -> Q_TO_STR_STATUS_Q1.IB
- AX_X_TO_BOOL_3.IN -> Q_TO_STR_STATUS_I3.IB
- Q_TO_BOOL_Q4.Q -> Q_TO_STR_STATUS_Q4.IB
- Q_TO_BOOL_Q2.Q -> Q_TO_STR_STATUS_Q2.IB
- AX_X_TO_BOOL_2.IN -> Q_TO_STR_STATUS_I2.IB
- AX_X_TO_BOOL_1.IN -> BOOL_TO_Q_I1.I
- BOOL_TO_Q_I1. -> Q_TO_STR_STATUS_I1.IB
- AX_X_TO_BOOL_2.IN -> BOOL_TO_Q_I2.I
- AX_X_TO_BOOL_4.IN -> BOOL_TO_Q_I4.I
- BOOL_TO_Q_I4. -> Q_TO_STR_STATUS_I4.IB
- AX_X_TO_BOOL_3.IN -> BOOL_TO_Q_I3.I
- ASSEMBLE_BYTE_FROM_QUARTERS. -> SPLIT_BYTE_INTO_QUARTERS.IN
- SPLIT_BYTE_INTO_QUARTERS.QUARTER_BYTE_00 -> Q_TO_BOOL_Q1.IB
- SPLIT_BYTE_INTO_QUARTERS.QUARTER_BYTE_03 -> Q_TO_BOOL_Q4.IB
- SPLIT_BYTE_INTO_QUARTERS.QUARTER_BYTE_01 -> Q_TO_BOOL_Q2.IB

- SPLIT_BYTE_INTO_QUARTERS.QUARTER_BYTE_02 -> Q_TO_BOOL_Q3.IB
- BOOL_TO_Q_I1. -> ASSEMBLE_BYTE_FROM_QUARTERS.QUARTER_BYTE_00
- BOOL_TO_Q_I2. -> ASSEMBLE_BYTE_FROM_QUARTERS.QUARTER_BYTE_01
- BOOL_TO_Q_I4. -> ASSEMBLE_BYTE_FROM_QUARTERS.QUARTER_BYTE_03
- BOOL_TO_Q_I3. -> ASSEMBLE_BYTE_FROM_QUARTERS.QUARTER_BYTE_02

Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_056_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.328 Uebung_057_AX: Reine Event-Schrittkette 5-Kanal (Adapter Version)

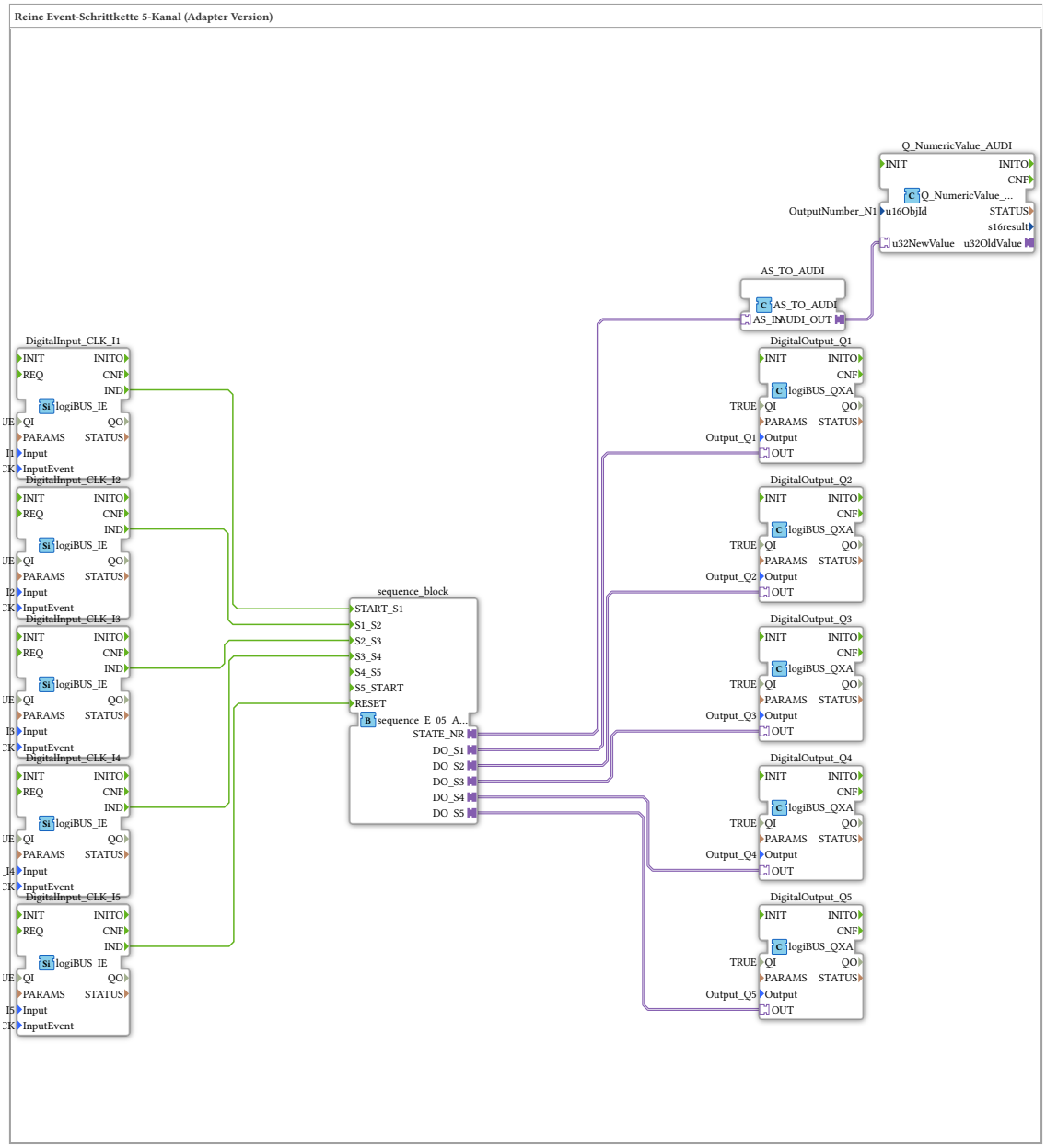
Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_057_AX`. Wir bauen eine rein ereignisgesteuerte Schrittkette mit AX-Adaptertechnologie.

Ziel der Übung

Realisierung einer Abfolge von 5 Ausgängen (Q1, Q2, Q3, Q4, Q5), bei der jeder Schritt ausschließlich durch ein externes Ereignis (Tasterdruck) weitergeschaltet wird - keine automatische Zeitsteuerung.

Beschreibung und Komponenten

Die Subapplikation `Uebung_057_AX` nutzt den Sequenzer `sequence_E_05_AX`, der 5 Zustände rein ereignisgesteuert durchläuft.



Jeder Übergang zwischen den Schritten wird durch eine eigene Tasterbetätigung (`I1` bis `I5`) ausgelöst - die Sequenz wartet an jedem Schritt, bis der nächste Taster gedrückt wird.

- **`Q_NumericValue_AUDI`**: Zeigt die aktuelle Schrittnummer auf dem ISOBUS-Terminal an.

Funktionsweise

1. Start durch Taster `I1` -> Sequenz springt zu `S1`, Ausgang `Q1` wird aktiv.
2. Jeder weitere Tasterdruck schaltet zum nächsten Schritt weiter.
3. Nach dem letzten Schritt bleibt die Sequenz stehen, bis sie zurückgesetzt wird.
4. Reset durch den letzten Taster -> Alles aus.

2.1.2.329 Übung 057: Reine Event-Schrittkette 5-Kanal

Thema: Ablaufsteuerungen in der Landtechnik

Situationsbeschreibung

Für ein System in der landwirtschaftlichen Automatisierung (z.B. eine Fassreinigung, eine Klappvorrichtung oder ein Gebläse) soll eine Ablaufsteuerung implementiert werden. Es werden 5 Stufen oder Aktoren nacheinander geschaltet.

Die Steuerung soll über den Baustein `sequence_E_05_AX` realisiert werden.

Funktionsbeschreibung der Ablaufsteuerung

- **Start:** Durch Betätigen des Tasters `I1` (`START_S1`).
- **Phasen:** 5 aufeinanderfolgende Ausgangsschritte (Ausgänge `Q1` bis `Q5`).
- **Übergänge:** Die Schrittübergänge laufen rein ereignisgesteuert ab (Taster I2, I3 etc.).
- **Reset:** Zu jedem Zeitpunkt kann die Schrittkette über einen Reset (z.B. Taster `I4` oder Softkey) in den Ausgangszustand (Schritt 0) zurückgesetzt werden.

Arbeitsauftrag

1. Öffnen Sie das 4diac-Projekt und legen Sie die SubApp `Uebung_057_AX` bzw. `Uebung_057` an.
2. Platzieren Sie den Sequenzer-Baustein `sequence_E_05_AX` und konfigurieren Sie die Parameter.
3. Verbinden Sie den Start-Taster `I1` und den Reset-Taster `I4` mit den entsprechenden Eingängen.
4. Schalten Sie die 5 Ausgänge auf die physischen Ausgänge `Output_Q1` bis `Output_Q5`.
5. Zur Visualisierung soll die Zustandsnummer `STATE_NR` auf dem ISOBUS Virtual Terminal-Ausgang `OutputNumber_N1` angezeigt werden.

2.1.2.330 Uebung_058_AX: Reine Event-Schrittkette 5-Kanal ENDLOS (Adapter Version)

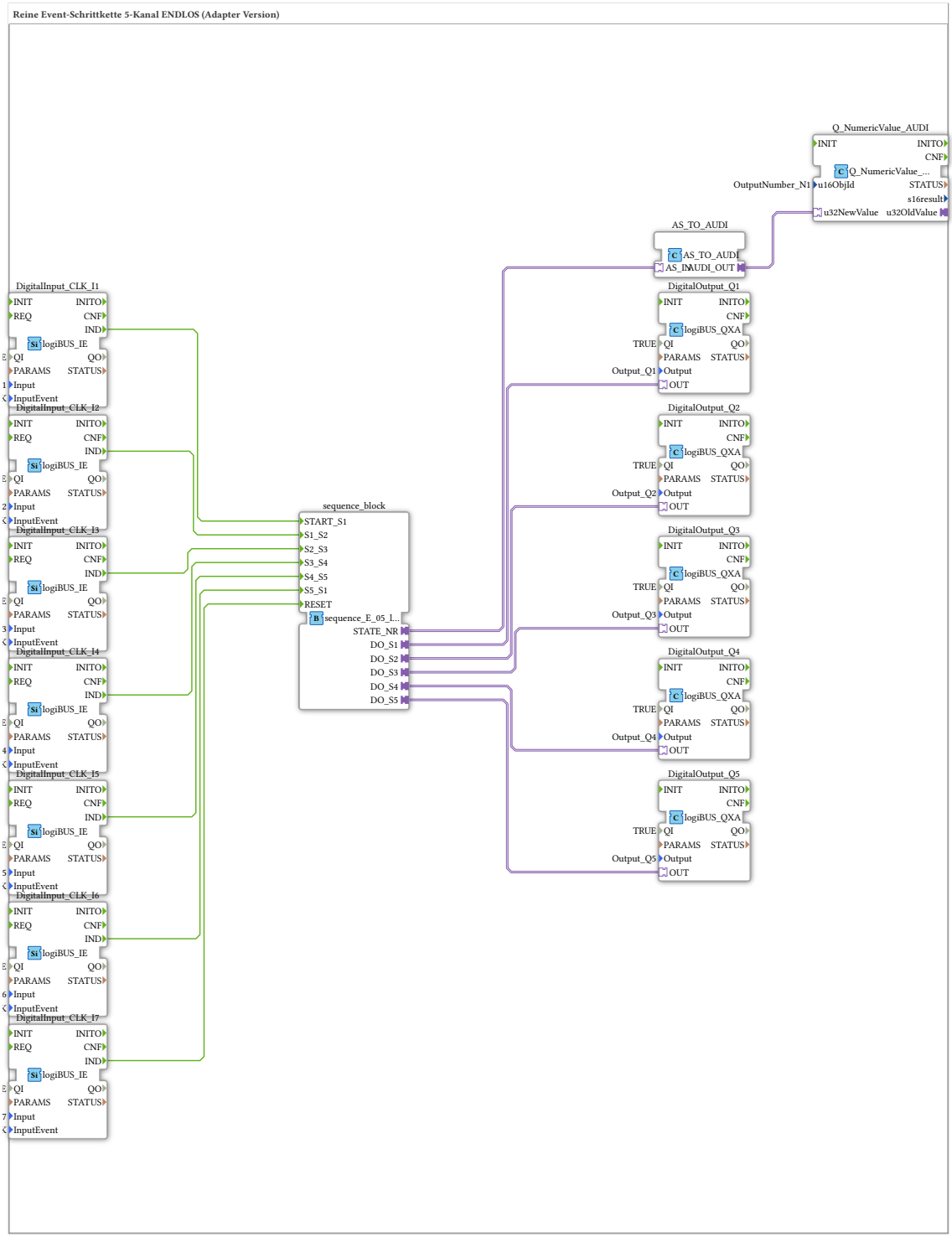
Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_058_AX`. Wir bauen eine rein ereignisgesteuerte Schrittkette mit AX-Adaptertechnologie.

Ziel der Übung

Realisierung einer Abfolge von 5 Ausgängen (Q1, Q2, Q3, Q4, Q5), bei der jeder Schritt ausschließlich durch ein externes Ereignis (Tasterdruck) weitergeschaltet wird - keine automatische Zeitsteuerung.

Beschreibung und Komponenten

Die Subapplikation `Uebung_058_AX` nutzt den Sequenzer `sequence_E_05_loop_AX`, der 5 Zustände rein ereignisgesteuert durchläuft.



Jeder Übergang zwischen den Schritten wird durch eine eigene Tasterbetätigung (`I1` bis `I7`) ausgelöst - die Sequenz wartet an jedem Schritt, bis der nächste Taster gedrückt wird.

- **`Q_NumericValue_AUDI`**: Zeigt die aktuelle Schrittnummer auf dem ISOBUS-Terminal an.

Funktionsweise

1. Start durch Taster `I1` -> Sequenz springt zu `S1`, Ausgang `Q1` wird aktiv.
2. Jeder weitere Tasterdruck schaltet zum nächsten Schritt weiter.
3. Nach dem letzten Schritt kehrt die Sequenz automatisch zu S1 zurück (ENDLOS).
4. Reset durch den letzten Taster -> Alles aus.

2.1.2.331 Übung 058: Reine Event-Schrittkette 5-Kanal ENDLOS

Thema: Ablaufsteuerungen in der Landtechnik

Situationsbeschreibung

Für ein System in der landwirtschaftlichen Automatisierung (z.B. eine Fassreinigung, eine Klappvorrichtung oder ein Gebläse) soll eine Ablaufsteuerung implementiert werden. Es werden 5 Stufen oder Aktoren nacheinander geschaltet.

Die Steuerung soll über den Baustein `sequence_E_05_loop_AX` realisiert werden.

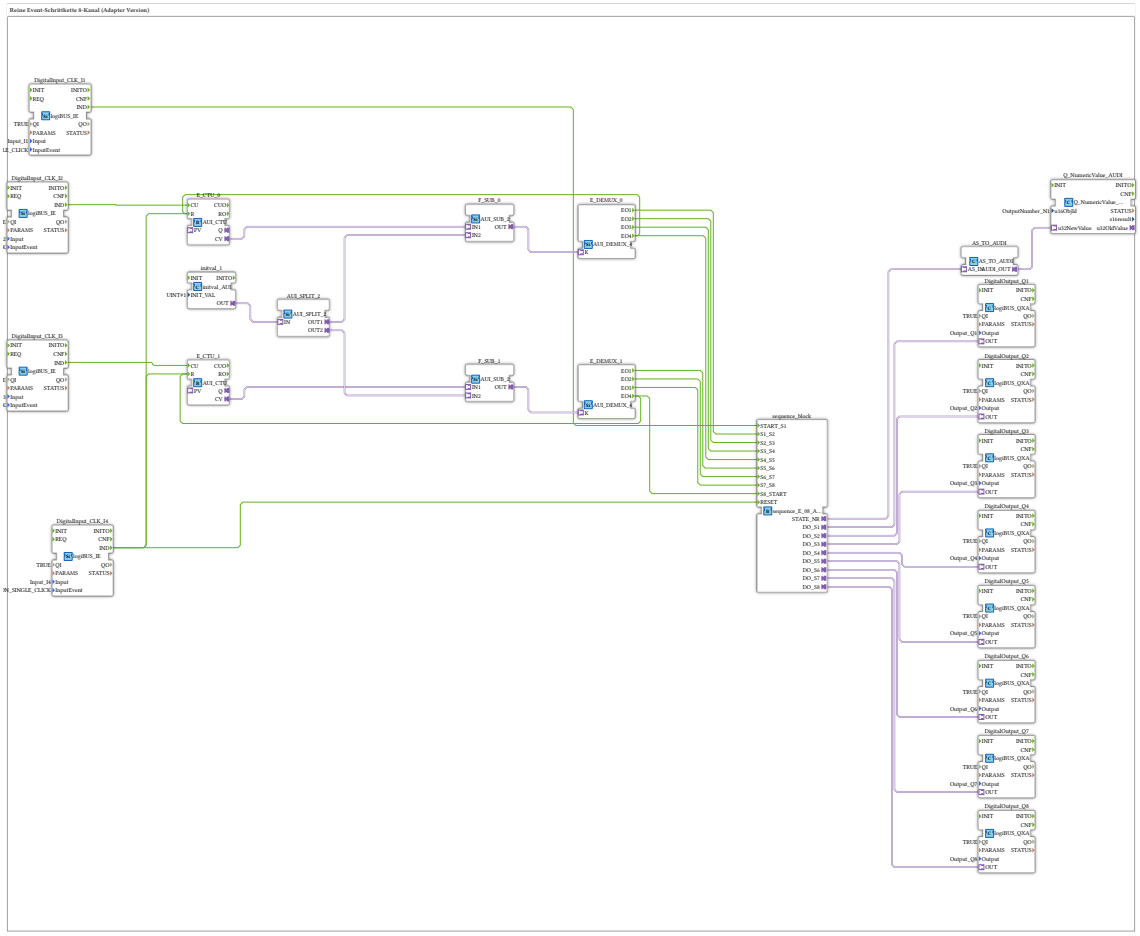
Funktionsbeschreibung der Ablaufsteuerung

- **Start:** Durch Betätigen des Tasters `I1` (`START_S1`).
- **Phasen:** 5 aufeinanderfolgende Ausgangsschritte (Ausgänge `Q1` bis `Q5`).
- **Übergänge:** Die Schrittübergänge laufen rein ereignisgesteuert ab (Taster I2, I3 etc.).
- **Reset:** Zu jedem Zeitpunkt kann die Schrittkette über einen Reset (z.B. Taster `I4` oder Softkey) in den Ausgangszustand (Schritt 0) zurückgesetzt werden.

Arbeitsauftrag

1. Öffnen Sie das 4diac-Projekt und legen Sie die SubApp `Uebung_058_AX` bzw. `Uebung_058` an.
2. Platzieren Sie den Sequenzer-Baustein `sequence_E_05_loop_AX` und konfigurieren Sie die Parameter.
3. Verbinden Sie den Start-Taster `I1` und den Reset-Taster `I4` mit den entsprechenden Eingängen.
4. Schalten Sie die 5 Ausgänge auf die physischen Ausgänge `Output_Q1` bis `Output_Q5`.
5. Zur Visualisierung soll die Zustandsnummer `STATE_NR` auf dem ISOBUS Virtual Terminal-Ausgang `OutputNumber_N1` angezeigt werden.

2.1.2.332 Uebung_059_AX: Reine Event-Schrittkette 8-Kanal (Adapter Version)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung realisiert eine rein ereignisgesteuerte Schrittkette mit AX-Adaptertechnologie: eine Abfolge von 8 Ausgängen (`Q1` bis `Q8`), bei der jeder Schritt ausschließlich durch ein externes Ereignis (Tasterklick) weitergeschaltet wird - keine automatische Zeitsteuerung. Da nur die physischen Taster `I1` bis `I4` zur Verfügung stehen, aber bis zu 8 Übergänge benötigt werden, zählt je ein Zähler die Klicks von `I2` bzw. `I3`, und ein nachgeschalteter Demultiplexer fächert den Zählerstand auf die einzelnen Übergänge des Sequenzers auf. So lassen sich mit zwei Tastern 8 Zustandsübergänge auslösen, ohne für jeden Schritt einen eigenen Taster zu benötigen.

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_CLK_I1** ... **DigitalInput_CLK_I4**: Klick-Eingänge
  - **Typ**: `logiBUS::io::DI::logiBUS_IE`
  - **Parameter**: `QI = TRUE`, `Input = Input_I1` ... `Input_I4`, jeweils `InputEvent = BUTTON_SINGLE_CLICK`
  - **Erklärung**: Melden einen Einzelklick auf `I1` bis `I4` als Ereignis `IND`.
- **sequence_block**: Sequenzer
  - **Typ**: `logiBUS::utils::sequence::event::sequence_E_08_AX`
  - **Parameter**: keine
  - **Erklärung**: Rein ereignisgesteuerte Sequenz mit 8 Zuständen. Die Ausgänge `DO_S1` ... `DO_S8` sind AX-Adapter, die Schrittnummer liegt am Adapter `STATE_NR` an. Die Eingänge `START_S1`, `S1_S2` ... `S7_S8`, `S8_START` und `RESET` steuern die Übergänge.
- **E_CTU_0**, **E_CTU_1**: Ereigniszähler
  - **Typ**: `adapter::events::unidirectional::AUI_CTU`
  - **Parameter**: keine
  - **Erklärung**: Zählt bei jedem Ereignis `CU` hoch und gibt den Zählerstand über den Adapter `CV` aus; `R` setzt den Zähler zurück.
- **initval_1**: Startwert 1
  - **Typ**: `adapter::types::unidirectional::AUI::initval::initval_AUI`
  - **Parameter**: `INIT_VAL = 1`
  - **Erklärung**: Stellt den konstanten Wert 1 als AUI-Signal bereit.
- **AUI_SPLIT_2**: Signalverteiler
  - **Typ**: `adapter::events::unidirectional::AUI_SPLIT_2`
  - **Parameter**: keine
  - **Erklärung**: Verteilt den Wert 1 von `initval_1` auf beide Subtrahierer.
- **F_SUB_0**, **F_SUB_1**: Subtrahierer
  - **Typ**: `adapter::iec61131::arithmetic::AUI_SUB_2`
  - **Parameter**: keine
  - **Erklärung**: Bilden `IN1 - IN2`, hier den Zählerstand minus 1. Damit entspricht der erste Klick dem Index 0.
- **E_DEMUX_0**, **E_DEMUX_1**: Ereignis-Demultiplexer
  - **Typ**: `adapter::events::unidirectional::AUI_DEMUX_4`
  - **Parameter**: keine
  - **Erklärung**: Leitet das Ereignis abhängig vom Index am Adapter `K` an `EO1` (Index 0), `EO2` (1), `EO3` (2) oder `EO4` (3) weiter.
- **AS_TO_AUDI**: Konvertierungsadapter
  - **Typ**: `adapter::conversion::unidirectional::AS_TO_AUDI`
  - **Parameter**: keine
  - **Erklärung**: Wandelt die Schrittnummer vom AS-Adapter in einen AUDI-Adapter.
- **Q_NumericValue_AUDI**: VT-Anzeige der Schrittnummer
  - **Typ**: `isobus::UT::Q::Q_NumericValue_AUDI`
  - **Parameter**: `u16ObjId = OutputNumber_N1`
  - **Erklärung**: Zeigt die aktuelle Schrittnummer auf dem ISOBUS-Terminal an.
- **DigitalOutput_Q1** ... **DigitalOutput_Q8**: Digitalausgänge
  - **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
  - **Parameter**: `QI = TRUE`, `Output = Output_Q1` ... `Output_Q8`
  - **Erklärung**: Geben den Zustand des jeweiligen Schritts an die physischen Ausgänge `Q1` bis `Q8` aus.

Programmablauf und Verbindungen

1. `DigitalInput_CLK_I1.IND` ⟶ `sequence_block.START_S1`: Ein Klick auf `I1` startet die Sequenz in Schritt 1.

2. `DigitalInput_CLK_I2.IND` ⟶ `E_CTU_0.CU`: Die Klicks auf `I2` werden gezählt. `E_CTU_0.CV` ⟶ `F_SUB_0.IN1` und `AUI_SPLIT_2.OUT1` ⟶ `F_SUB_0.IN2` (Wert 1), `F_SUB_0.OUT` ⟶ `E_DEMUX_0.K`: Der Index ist Zählerstand minus 1.
3. `E_DEMUX_0.EO1` ⟶ `sequence_block.S1_S2`, `E_DEMUX_0.EO2` ⟶ `S2_S3`, `E_DEMUX_0.EO3` ⟶ `S3_S4`, `E_DEMUX_0.EO4` ⟶ `S4_S5`: Der 1. bis 4. Klick auf `I2` schaltet von Schritt 1 nach 2, 2 nach 3, 3 nach 4 bzw. 4 nach 5.
4. `E_DEMUX_0.EO4` ⟶ `E_CTU_0.R`: Nach dem 4. Klick wird der Zähler `E_CTU_0` zurückgesetzt.
5. `DigitalInput_CLK_I3.IND` ⟶ `E_CTU_1.CU`, `E_CTU_1.CV` ⟶ `F_SUB_1.IN1`, `AUI_SPLIT_2.OUT2` ⟶ `F_SUB_1.IN2`, `F_SUB_1.OUT` ⟶ `E_DEMUX_1.K`: Gleiches Prinzip für die Klicks auf `I3`.
6. `E_DEMUX_1.EO1` ⟶ `sequence_block.S5_S6`, `EO2` ⟶ `S6_S7`, `EO3` ⟶ `S7_S8`, `EO4` ⟶ `S8_START`: Der 1. bis 4. Klick auf `I3` schaltet von Schritt 5 nach 6, 6 nach 7, 7 nach 8 bzw. 8 zurück zum Start. `E_DEMUX_1.EO4` ⟶ `E_CTU_1.R` setzt auch diesen Zähler zurück.
7. `DigitalInput_CLK_I4.IND` ⟶ `sequence_block.RESET`, `E_CTU_0.R` und `E_CTU_1.R`: Ein Klick auf `I4` setzt Sequenz und beide Zähler zurück.
8. `sequence_block.DO_S1` … `DO_S8` ⟶ `DigitalOutput_Q1.OUT` … `DigitalOutput_Q8.OUT` (AdapterConnections): Der aktive Schritt schaltet den zugehörigen Ausgang.
9. `sequence_block.STATE_NR` ⟶ `AS_TO_AUDI.AS_IN` und `AS_TO_AUDI.AUDI_OUT` ⟶ `Q_NumericValue_AUDI.u32NewValue`: Die Schrittnummer wird auf dem VT angezeigt.

Verhalten: Klick auf `I1` ⟶ Schritt 1, `Q1` aktiv. Jeder weitere Klick auf `I2` (Schritte 1 bis 5) bzw. `I3` (Schritte 5 bis 8 und zurück zum Start) schaltet zum nächsten Schritt weiter. Ein Klick auf `I4` setzt alles zurück.

Zusammenfassung

Die Übung zeigt eine rein ereignisgesteuerte 8-Schritt-Kette in AX-Adaptertechnik. Mit je einem Zähler und einem Demultiplexer pro Taster (`I2`, `I3`) werden 8 Übergänge mit nur zwei Weiterschalt-Tastern realisiert; `I1` startet und `I4` setzt zurück.

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

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

2.1.2.333 Übung 059: Reine Event-Schrittkette 8-Kanal

Thema: Ablaufsteuerungen in der Landtechnik

Situationsbeschreibung

Für ein System in der landwirtschaftlichen Automatisierung (z.B. eine Fassreinigung, eine Klappvorrichtung oder ein Gebläse) soll eine Ablaufsteuerung implementiert werden. Es werden 8 Stufen oder Aktoren nacheinander geschaltet.

Die Steuerung soll über den Baustein `sequence_E_08_AX` realisiert werden.

Funktionsbeschreibung der Ablaufsteuerung

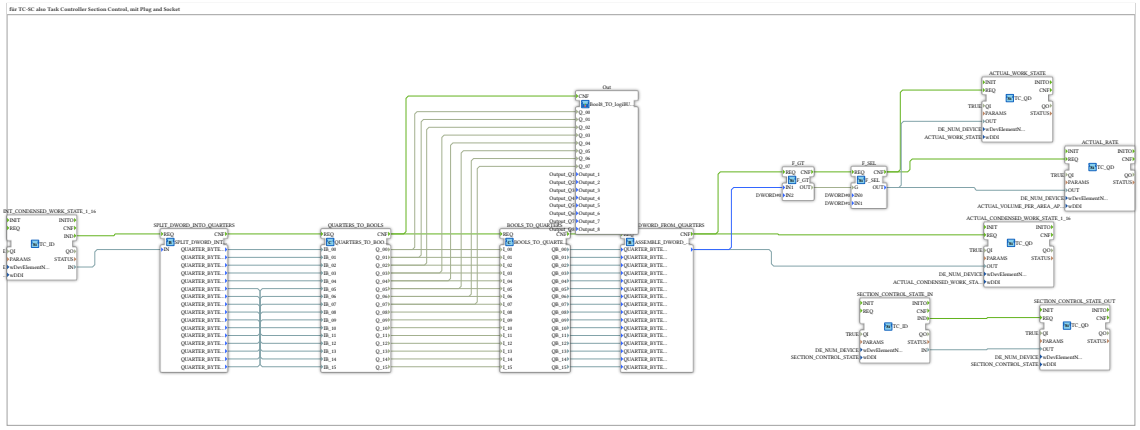
- **Start:** Durch Betätigen des Tasters `I1` (`START_S1`).
- **Phasen:** 8 aufeinanderfolgende Ausgangsschritte (Ausgänge `Q1` bis `Q8`).
- **Übergänge:** Die Schrittübergänge laufen rein ereignisgesteuert ab (Taster I2, I3 etc.).
- **Reset:** Zu jedem Zeitpunkt kann die Schrittkette über einen Reset (z.B. Taster `I4` oder Softkey) in den Ausgangszustand (Schritt 0) zurückgesetzt werden.

Arbeitsauftrag

1. Öffnen Sie das 4diac-Projekt und legen Sie die SubApp `Uebung_059_AX` bzw. `Uebung_059` an.
2. Platzieren Sie den Sequenzer-Baustein `sequence_E_08_AX` und konfigurieren Sie die Parameter.
3. Verbinden Sie den Start-Taster `I1` und den Reset-Taster `I4` mit den entsprechenden Eingängen.
4. Schalten Sie die 8 Ausgänge auf die physischen Ausgänge `Output_Q1` bis `Output_Q8`.
5. Zur Visualisierung soll die Zustandsnummer `STATE_NR` auf dem ISOBUS Virtual Terminal-Ausgang `OutputNumber_N1` angezeigt werden.

2.1.2.334 Uebung_060_AX: für TC-SC also Task Controller Section Control, mit Plug and Socket

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_060_AX (für TC-SC also Task Controller Section Control, mit Plug and Socket).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **für TC-SC also Task Controller Section Control, mit Plug and Socket**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_060_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **SETPOINT_CONDENSED_WORK_STATE_1_16**: Instanz des Typs isobus::TC::io::TC_ID.
  - Parameter QI = TRUE
  - Parameter wDevElementNumb = DE_NUM_DEVICE
  - Parameter wDDI = SETPOINT_CONDENSED_WORK_STATE_1_16
- **SPLIT_DWORD_INTO_QUARTERS**: Instanz des Typs eclipse4diac::utils::splitting::SPLIT_DWORD_INTO_QUARTERS.
- **ASSEMBLE_DWORD_FROM_QUARTERS**: Instanz des Typs eclipse4diac::utils::assembling::ASSEMBLE_DWORD_FROM_QUARTERS.
- **ACTUAL_CONDENSED_WORK_STATE_1_16**: Instanz des Typs isobus::TC::io::TC_QD.
  - Parameter QI = TRUE
  - Parameter wDevElementNumb = DE_NUM_DEVICE
  - Parameter wDDI = ACTUAL_CONDENSED_WORK_STATE_1_16
- **BOOLS_TO_QUARTERS**: Instanz des Typs logiBUS::utils::quarter::BOOLS_TO_QUARTERS.
- **QUARTERS_TO_BOOLS**: Instanz des Typs logiBUS::utils::quarter::QUARTERS_TO_BOOLS.
- **SECTION_CONTROL_STATE_IN**: Instanz des Typs isobus::TC::io::TC_ID.
  - Parameter QI = TRUE
  - Parameter wDevElementNumb = DE_NUM_DEVICE
  - Parameter wDDI = SECTION_CONTROL_STATE
- **SECTION_CONTROL_STATE_OUT**: Instanz des Typs isobus::TC::io::TC_QD.
  - Parameter QI = TRUE
  - Parameter wDevElementNumb = DE_NUM_DEVICE
  - Parameter wDDI = SECTION_CONTROL_STATE
- **ACTUAL_RATE**: Instanz des Typs isobus::TC::io::TC_QD.
  - Parameter QI = TRUE
  - Parameter wDevElementNumb = DE_NUM_DEVICE
  - Parameter wDDI = ACTUAL_VOLUME_PER_AREA_APPLICATION_RATE
- **ACTUAL_WORK_STATE**: Instanz des Typs isobus::TC::io::TC_QD.
  - Parameter QI = TRUE
  - Parameter wDevElementNumb = DE_NUM_DEVICE
  - Parameter wDDI = ACTUAL_WORK_STATE
- **F_SEL**: Instanz des Typs iec61131::selection::F_SEL.
  - Parameter IN0 = DWORD#0
  - Parameter IN1 = DWORD#1
- **F_GT**: Instanz des Typs iec61131::comparison::F_GT.

- Parameter IN2 = DWORD#0

Verbindungen und Schnittstellen

Ereignisverbindungen:

- SETPOINT_CONDENSED_WORK_STATE_1_16.IND -> SPLIT_DWORD_INTO_QUARTERS.REQ
- ASSEMBLE_DWORD_FROM_QUARTERS.CNF -> ACTUAL_CONDENSED_WORK_STATE_1_16.REQ
- BOOLS_TO_QUARTERS.CNF -> ASSEMBLE_DWORD_FROM_QUARTERS.REQ
- SPLIT_DWORD_INTO_QUARTERS.CNF -> QUARTERS_TO_BOOLS.REQ
- QUARTERS_TO_BOOLS.CNF -> BOOLS_TO_QUARTERS.REQ
- QUARTERS_TO_BOOLS.CNF -> Out.CNF
- SECTION_CONTROL_STATE_IN.IND -> SECTION_CONTROL_STATE_OUT.REQ
- F_GT.CNF -> F_SEL.REQ
- ASSEMBLE_DWORD_FROM_QUARTERS.CNF -> F_GT.REQ
- F_SEL.CNF -> ACTUAL_WORK_STATE.REQ
- F_SEL.CNF -> ACTUAL_RATE.REQ

Datenverbindungen:

- SETPOINT_CONDENSED_WORK_STATE_1_16.IN -> SPLIT_DWORD_INTO_QUARTERS.IN
- QUARTERS_TO_BOOLS.Q_00 -> BOOLS_TO_QUARTERS.I_00
- QUARTERS_TO_BOOLS.Q_02 -> BOOLS_TO_QUARTERS.I_02
- QUARTERS_TO_BOOLS.Q_01 -> BOOLS_TO_QUARTERS.I_01
- QUARTERS_TO_BOOLS.Q_03 -> BOOLS_TO_QUARTERS.I_03
- QUARTERS_TO_BOOLS.Q_04 -> BOOLS_TO_QUARTERS.I_04
- QUARTERS_TO_BOOLS.Q_05 -> BOOLS_TO_QUARTERS.I_05
- QUARTERS_TO_BOOLS.Q_06 -> BOOLS_TO_QUARTERS.I_06
- QUARTERS_TO_BOOLS.Q_07 -> BOOLS_TO_QUARTERS.I_07
- QUARTERS_TO_BOOLS.Q_08 -> BOOLS_TO_QUARTERS.I_08
- QUARTERS_TO_BOOLS.Q_09 -> BOOLS_TO_QUARTERS.I_09
- QUARTERS_TO_BOOLS.Q_10 -> BOOLS_TO_QUARTERS.I_10
- QUARTERS_TO_BOOLS.Q_11 -> BOOLS_TO_QUARTERS.I_11
- QUARTERS_TO_BOOLS.Q_12 -> BOOLS_TO_QUARTERS.I_12
- QUARTERS_TO_BOOLS.Q_13 -> BOOLS_TO_QUARTERS.I_13
- QUARTERS_TO_BOOLS.Q_14 -> BOOLS_TO_QUARTERS.I_14
- QUARTERS_TO_BOOLS.Q_15 -> BOOLS_TO_QUARTERS.I_15
- QUARTERS_TO_BOOLS.Q_06 -> Out.Q_06
- QUARTERS_TO_BOOLS.Q_00 -> Out.Q_00
- QUARTERS_TO_BOOLS.Q_04 -> Out.Q_04
- QUARTERS_TO_BOOLS.Q_01 -> Out.Q_01
- QUARTERS_TO_BOOLS.Q_05 -> Out.Q_05
- QUARTERS_TO_BOOLS.Q_07 -> Out.Q_07
- QUARTERS_TO_BOOLS.Q_02 -> Out.Q_02
- QUARTERS_TO_BOOLS.Q_03 -> Out.Q_03
- BOOLS_TO_QUARTERS.QB_00 -> ASSEMBLE_DWORD_FROM_QUARTERS.QUARTER_BYTE_00
- BOOLS_TO_QUARTERS.QB_01 -> ASSEMBLE_DWORD_FROM_QUARTERS.QUARTER_BYTE_01
- BOOLS_TO_QUARTERS.QB_02 -> ASSEMBLE_DWORD_FROM_QUARTERS.QUARTER_BYTE_02
- BOOLS_TO_QUARTERS.QB_03 -> ASSEMBLE_DWORD_FROM_QUARTERS.QUARTER_BYTE_03
- ASSEMBLE_DWORD_FROM_QUARTERS. -> ACTUAL_CONDENSED_WORK_STATE_1_16.OUT
- BOOLS_TO_QUARTERS.QB_04 -> ASSEMBLE_DWORD_FROM_QUARTERS.QUARTER_BYTE_04
- BOOLS_TO_QUARTERS.QB_05 -> ASSEMBLE_DWORD_FROM_QUARTERS.QUARTER_BYTE_05
- BOOLS_TO_QUARTERS.QB_06 -> ASSEMBLE_DWORD_FROM_QUARTERS.QUARTER_BYTE_06
- BOOLS_TO_QUARTERS.QB_07 -> ASSEMBLE_DWORD_FROM_QUARTERS.QUARTER_BYTE_07
- BOOLS_TO_QUARTERS.QB_08 -> ASSEMBLE_DWORD_FROM_QUARTERS.QUARTER_BYTE_08
- BOOLS_TO_QUARTERS.QB_09 -> ASSEMBLE_DWORD_FROM_QUARTERS.QUARTER_BYTE_09
- BOOLS_TO_QUARTERS.QB_10 -> ASSEMBLE_DWORD_FROM_QUARTERS.QUARTER_BYTE_10
- BOOLS_TO_QUARTERS.QB_11 -> ASSEMBLE_DWORD_FROM_QUARTERS.QUARTER_BYTE_11
- BOOLS_TO_QUARTERS.QB_12 -> ASSEMBLE_DWORD_FROM_QUARTERS.QUARTER_BYTE_12
- BOOLS_TO_QUARTERS.QB_13 -> ASSEMBLE_DWORD_FROM_QUARTERS.QUARTER_BYTE_13
- BOOLS_TO_QUARTERS.QB_14 -> ASSEMBLE_DWORD_FROM_QUARTERS.QUARTER_BYTE_14
- BOOLS_TO_QUARTERS.QB_15 -> ASSEMBLE_DWORD_FROM_QUARTERS.QUARTER_BYTE_15

- SPLIT_DWORD_INTO_QUARTERS.QUARTER_BYTE_00 -> QUARTERS_TO_BOOLS.IB_00
- SPLIT_DWORD_INTO_QUARTERS.QUARTER_BYTE_01 -> QUARTERS_TO_BOOLS.IB_01
- SPLIT_DWORD_INTO_QUARTERS.QUARTER_BYTE_02 -> QUARTERS_TO_BOOLS.IB_02
- SPLIT_DWORD_INTO_QUARTERS.QUARTER_BYTE_03 -> QUARTERS_TO_BOOLS.IB_03
- SPLIT_DWORD_INTO_QUARTERS.QUARTER_BYTE_04 -> QUARTERS_TO_BOOLS.IB_04
- SPLIT_DWORD_INTO_QUARTERS.QUARTER_BYTE_15 -> QUARTERS_TO_BOOLS.IB_15
- SPLIT_DWORD_INTO_QUARTERS.QUARTER_BYTE_14 -> QUARTERS_TO_BOOLS.IB_14
- SPLIT_DWORD_INTO_QUARTERS.QUARTER_BYTE_13 -> QUARTERS_TO_BOOLS.IB_13
- SPLIT_DWORD_INTO_QUARTERS.QUARTER_BYTE_12 -> QUARTERS_TO_BOOLS.IB_12
- SPLIT_DWORD_INTO_QUARTERS.QUARTER_BYTE_11 -> QUARTERS_TO_BOOLS.IB_11
- SPLIT_DWORD_INTO_QUARTERS.QUARTER_BYTE_10 -> QUARTERS_TO_BOOLS.IB_10
- SPLIT_DWORD_INTO_QUARTERS.QUARTER_BYTE_09 -> QUARTERS_TO_BOOLS.IB_09
- SPLIT_DWORD_INTO_QUARTERS.QUARTER_BYTE_08 -> QUARTERS_TO_BOOLS.IB_08
- SPLIT_DWORD_INTO_QUARTERS.QUARTER_BYTE_07 -> QUARTERS_TO_BOOLS.IB_07
- SPLIT_DWORD_INTO_QUARTERS.QUARTER_BYTE_06 -> QUARTERS_TO_BOOLS.IB_06
- SPLIT_DWORD_INTO_QUARTERS.QUARTER_BYTE_05 -> QUARTERS_TO_BOOLS.IB_05
- SECTION_CONTROL_STATE_IN.IN -> SECTION_CONTROL_STATE_OUT.OUT
- F_GT.OUT -> F_SEL.G
- ASSEMBLE_DWORD_FROM_QUARTERS. -> F_GT.IN1
- F_SEL.OUT -> ACTUAL_RATE.OUT
- F_SEL.OUT -> ACTUAL_WORK_STATE.OUT

Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_060_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.335 Uebung_061_AX: Reine Zeitsteuerung 4-Kanal (Adapter Version)

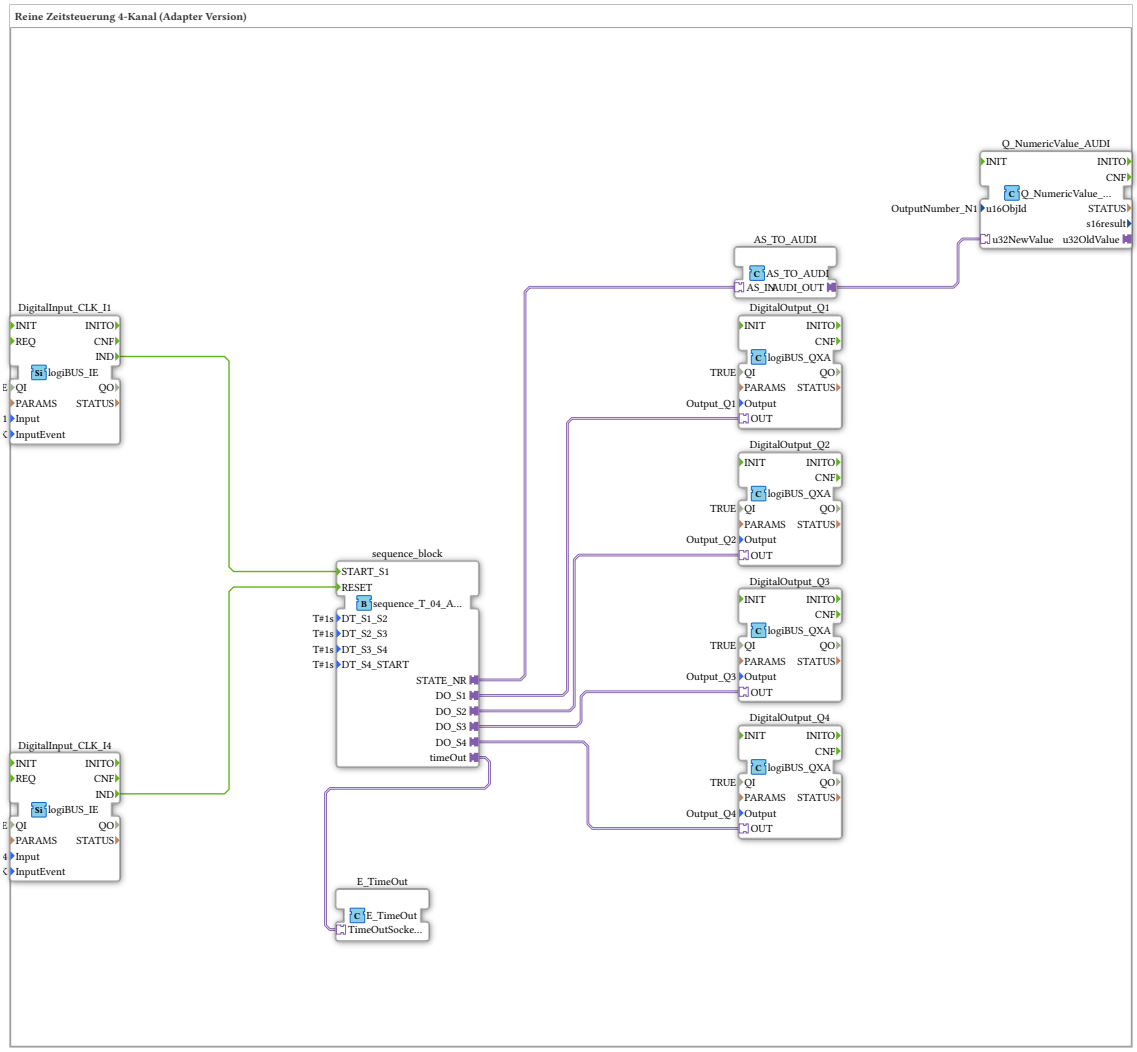
Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_061_AX`. Wir bauen eine rein zeitgesteuerte Schrittkette mit AX-Adaptertechnologie.

Ziel der Übung

Realisierung einer automatischen Abfolge von 4 Ausgängen (Q1, Q2, Q3, Q4), bei der jeder Schritt ausschließlich nach Ablauf einer festen Wartezeit (`DT_S1_S2` usw., je `T#1s`) zum nächsten wechselt - ganz ohne weitere Tasterbetätigung während des Ablaufs.

Beschreibung und Komponenten

Die Subapplikation `Uebung_061_AX` nutzt den Sequenzer `sequence_T_04_AX` (AX-Adaptervariante), der 4 Zustände (`S1` bis `S4`) rein zeitgesteuert durchläuft.



- **`sequence_T_04_AX`**: Verwaltet die 4 Schritte, jeder Übergang erfolgt nach Ablauf der jeweiligen `DT_Sx_Sy`-Parameter.
- **`E_TimeOut`**: Überwacht die Sequenz als Watchdog.
- **`Q_NumericValue_AUDI`**: Zeigt die aktuelle Schrittnummer auf dem ISOBUS-Terminal an.

Funktionsweise

1. Start durch Taster `I1` -> Sequenz springt zu `S1`.
2. Jeder Ausgang (`Q1, Q2, Q3, Q4`) wird für die eingestellte Zeit aktiv geschaltet, bevor automatisch zum nächsten Schritt gewechselt wird.
3. Nach dem letzten Schritt endet die Sequenz.
4. Reset durch Taster `I4` -> Alles aus, Sequenz zurück in den Grundzustand.

2.1.2.336 Übung 061: Reine Zeitsteuerung 4-Kanal

Thema: Ablaufsteuerungen in der Landtechnik

Situationsbeschreibung

Für ein System in der landwirtschaftlichen Automatisierung (z.B. eine Fassreinigung, eine Klappvorrichtung oder ein Gebläse) soll eine Ablaufsteuerung implementiert werden. Es werden 4 Stufen oder Aktoren nacheinander geschaltet.

Die Steuerung soll über den Baustein `sequence_T_04_AX` realisiert werden.

Funktionsbeschreibung der Ablaufsteuerung

- **Start:** Durch Betätigen des Tasters `I1` (`START_S1`).

- **Phasen:** 4 aufeinanderfolgende Ausgangsschritte (Ausgänge `Q1` bis `Q4`).
- **Übergänge:** Die Schrittübergänge laufen rein zeitgesteuert ab (jeder Schritt dauert 1 Sekunde).
- **Reset:** Zu jedem Zeitpunkt kann die Schrittkette über einen Reset (z.B. Taster `I4` oder Softkey) in den Ausgangszustand (Schritt 0) zurückgesetzt werden.

Arbeitsauftrag

1. Öffnen Sie das 4diac-Projekt und legen Sie die SubApp `Uebung_061_AX` bzw. `Uebung_061` an.
2. Platzieren Sie den Sequenzer-Baustein `sequence_T_04_AX` und konfigurieren Sie die Parameter.
3. Verbinden Sie den Start-Taster `I1` und den Reset-Taster `I4` mit den entsprechenden Eingängen.
4. Schalten Sie die 4 Ausgänge auf die physischen Ausgänge `Output_Q1` bis `Output_Q4`.
5. Zur Visualisierung soll die Zustandsnummer `STATE_NR` auf dem ISOBUS Virtual Terminal-Ausgang `OutputNumber_N1` angezeigt werden.

2.1.2.337 Uebung_062_AX: Reine Zeitsteuerung 5-Kanal (Adapter Version)

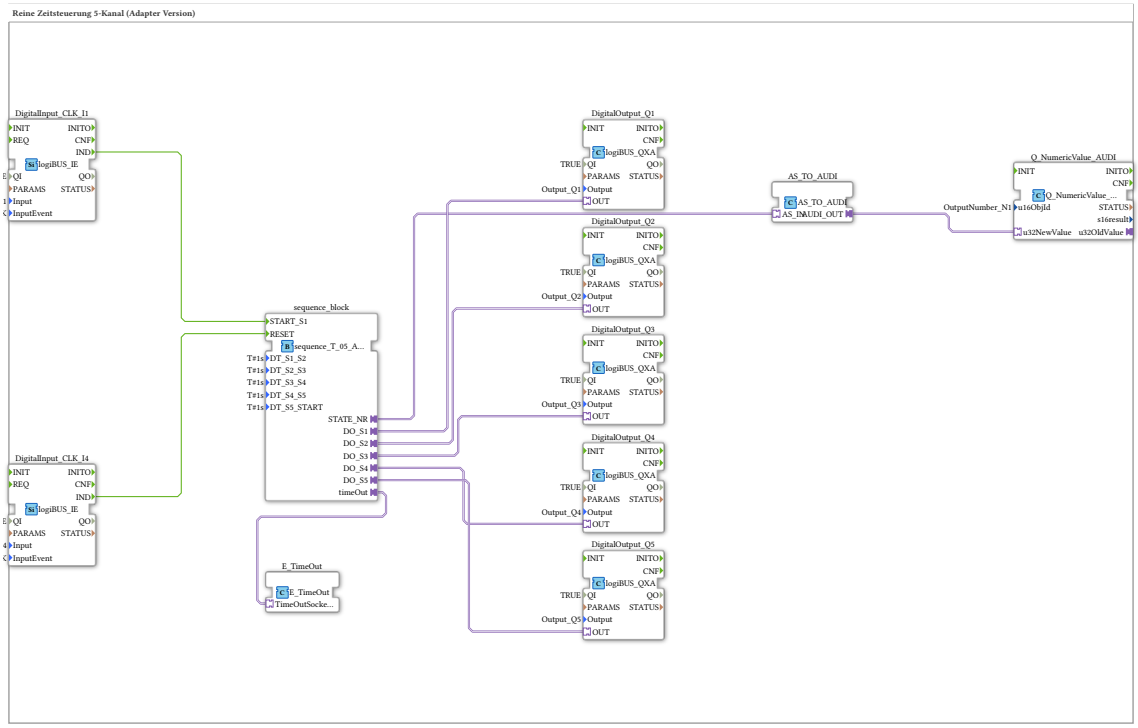
Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_062_AX`. Wir bauen eine rein zeitgesteuerte Schrittkette mit AX-Adaptertechnologie.

Ziel der Übung

Realisierung einer automatischen Abfolge von 5 Ausgängen (Q1, Q2, Q3, Q4, Q5), bei der jeder Schritt ausschließlich nach Ablauf einer festen Wartezeit (`DT_S1_S2` usw., je `T#1s`) zum nächsten wechselt - ganz ohne weitere Tasterbetätigung während des Ablaufs.

Beschreibung und Komponenten

Die Subapplikation `Uebung_062_AX` nutzt den Sequenzer `sequence_T_05_AX` (AX-Adaptervariante), der 5 Zustände (`S1` bis `S5`) rein zeitgesteuert durchläuft.



- **`sequence_T_05_AX`**: Verwaltet die 5 Schritte, jeder Übergang erfolgt nach Ablauf der jeweiligen `DT_Sx_Sy`-Parameter.
- **`E_TimeOut`**: Überwacht die Sequenz als Watchdog.
- **`Q_NumericValue_AUDI`**: Zeigt die aktuelle Schrittnummer auf dem ISOBUS-Terminal an.

Funktionsweise

1. Start durch Taster `I1` -> Sequenz springt zu `S1`.

2. Jeder Ausgang (`Q1, Q2, Q3, Q4, Q5`) wird für die eingestellte Zeit aktiv geschaltet, bevor automatisch zum nächsten Schritt gewechselt wird.
3. Nach dem letzten Schritt endet die Sequenz.
4. Reset durch Taster `I4` -> Alles aus, Sequenz zurück in den Grundzustand.

2.1.2.338 Übung 062: Reine Zeitsteuerung 5-Kanal

Thema: Ablaufsteuerungen in der Landtechnik

Situationsbeschreibung

Für ein System in der landwirtschaftlichen Automatisierung (z.B. eine Fassreinigung, eine Klappvorrichtung oder ein Gebläse) soll eine Ablaufsteuerung implementiert werden. Es werden 5 Stufen oder Aktoren nacheinander geschaltet.

Die Steuerung soll über den Baustein `sequence_T_05_AX` realisiert werden.

Funktionsbeschreibung der Ablaufsteuerung

- **Start:** Durch Betätigen des Tasters `I1` (`START_S1`).
- **Phasen:** 5 aufeinanderfolgende Ausgangsschritte (Ausgänge `Q1` bis `Q5`).
- **Übergänge:** Die Schrittübergänge laufen rein zeitgesteuert ab (jeder Schritt dauert 1 Sekunde).
- **Reset:** Zu jedem Zeitpunkt kann die Schrittkette über einen Reset (z.B. Taster `I4` oder Softkey) in den Ausgangszustand (Schritt 0) zurückgesetzt werden.

Arbeitsauftrag

1. Öffnen Sie das 4diac-Projekt und legen Sie die SubApp `Uebung_062_AX` bzw. `Uebung_062` an.
2. Platzieren Sie den Sequenzer-Baustein `sequence_T_05_AX` und konfigurieren Sie die Parameter.
3. Verbinden Sie den Start-Taster `I1` und den Reset-Taster `I4` mit den entsprechenden Eingängen.
4. Schalten Sie die 5 Ausgänge auf die physischen Ausgänge `Output_Q1` bis `Output_Q5`.
5. Zur Visualisierung soll die Zustandsnummer `STATE_NR` auf dem ISOBUS Virtual Terminal-Ausgang `OutputNumber_N1` angezeigt werden.

2.1.2.339 Uebung_063_AX: Reine Zeitsteuerung 8-Kanal (Adapter Version)

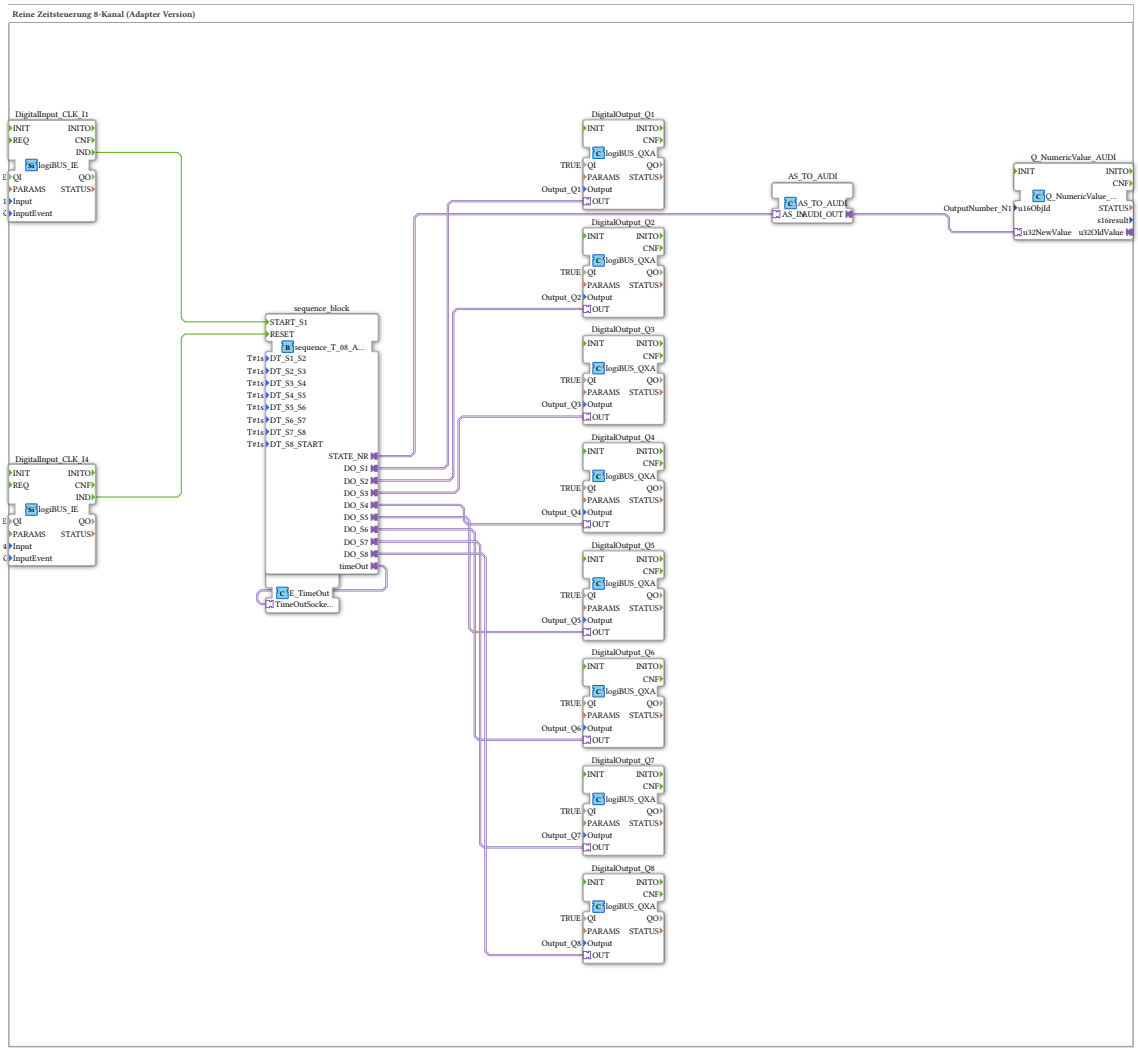
Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_063_AX`. Wir bauen eine rein zeitgesteuerte Schrittkette mit AX-Adaptertechnologie.

Ziel der Übung

Realisierung einer automatischen Abfolge von 8 Ausgängen (Q1, Q2, Q3, Q4, Q5, Q6, Q7, Q8), bei der jeder Schritt ausschließlich nach Ablauf einer festen Wartezeit (`DT_S1_S2` usw., je `T#1s`) zum nächsten wechselt - ganz ohne weitere Tasterbetätigung während des Ablaufs.

Beschreibung und Komponenten

Die Subapplikation `Uebung_063_AX` nutzt den Sequenzer `sequence_T_08_AX` (AX-Adaptervariante), der 8 Zustände (`S1` bis `S8`) rein zeitgesteuert durchläuft.



- **`sequence_T_08_AX`**: Verwaltet die 8 Schritte, jeder Übergang erfolgt nach Ablauf der jeweiligen `DT_Sx_Sy`-Parameter.
- **`E_TimeOut`**: Überwacht die Sequenz als Watchdog.
- **`Q_NumericValue_AUDI`**: Zeigt die aktuelle Schrittnummer auf dem ISOBUS-Terminal an.

Funktionsweise

1. Start durch Taster `I1` -> Sequenz springt zu `S1`.
2. Jeder Ausgang (`Q1, Q2, Q3, Q4, Q5, Q6, Q7, Q8`) wird für die eingestellte Zeit aktiv geschaltet, bevor automatisch zum nächsten Schritt gewechselt wird.
3. Nach dem letzten Schritt endet die Sequenz.
4. Reset durch Taster `I4` -> Alles aus, Sequenz zurück in den Grundzustand.

2.1.2.340 Übung 063: Reine Zeitsteuerung 8-Kanal

Thema: Ablaufsteuerungen in der Landtechnik

Situationsbeschreibung

Für ein System in der landwirtschaftlichen Automatisierung (z.B. eine Fassreinigung, eine Klappvorrichtung oder ein Gebläse) soll eine Ablaufsteuerung implementiert werden. Es werden 8 Stufen oder Aktoren nacheinander geschaltet.

Die Steuerung soll über den Baustein `sequence_T_08_AX` realisiert werden.

Funktionsbeschreibung der Ablaufsteuerung

- **Start:** Durch Betätigen des Tasters `I1` (`START_S1`).

- **Phasen:** 8 aufeinanderfolgende Ausgangsschritte (Ausgänge `Q1` bis `Q8`).
- **Übergänge:** Die Schrittübergänge laufen rein zeitgesteuert ab (jeder Schritt dauert 1 Sekunde).
- **Reset:** Zu jedem Zeitpunkt kann die Schrittkette über einen Reset (z.B. Taster `I4` oder Softkey) in den Ausgangszustand (Schritt 0) zurückgesetzt werden.

Arbeitsauftrag

1. Öffnen Sie das 4diac-Projekt und legen Sie die SubApp `Uebung_063_AX` bzw. `Uebung_063` an.
2. Platzieren Sie den Sequenzer-Baustein `sequence_T_08_AX` und konfigurieren Sie die Parameter.
3. Verbinden Sie den Start-Taster `I1` und den Reset-Taster `I4` mit den entsprechenden Eingängen.
4. Schalten Sie die 8 Ausgänge auf die physischen Ausgänge `Output_Q1` bis `Output_Q8`.
5. Zur Visualisierung soll die Zustandsnummer `STATE_NR` auf dem ISOBUS Virtual Terminal-Ausgang `OutputNumber_N1` angezeigt werden.

2.1.2.341 Uebung_064_AX: Musterschrittkette 8-Kanal ENDLOS (Adapter Version)

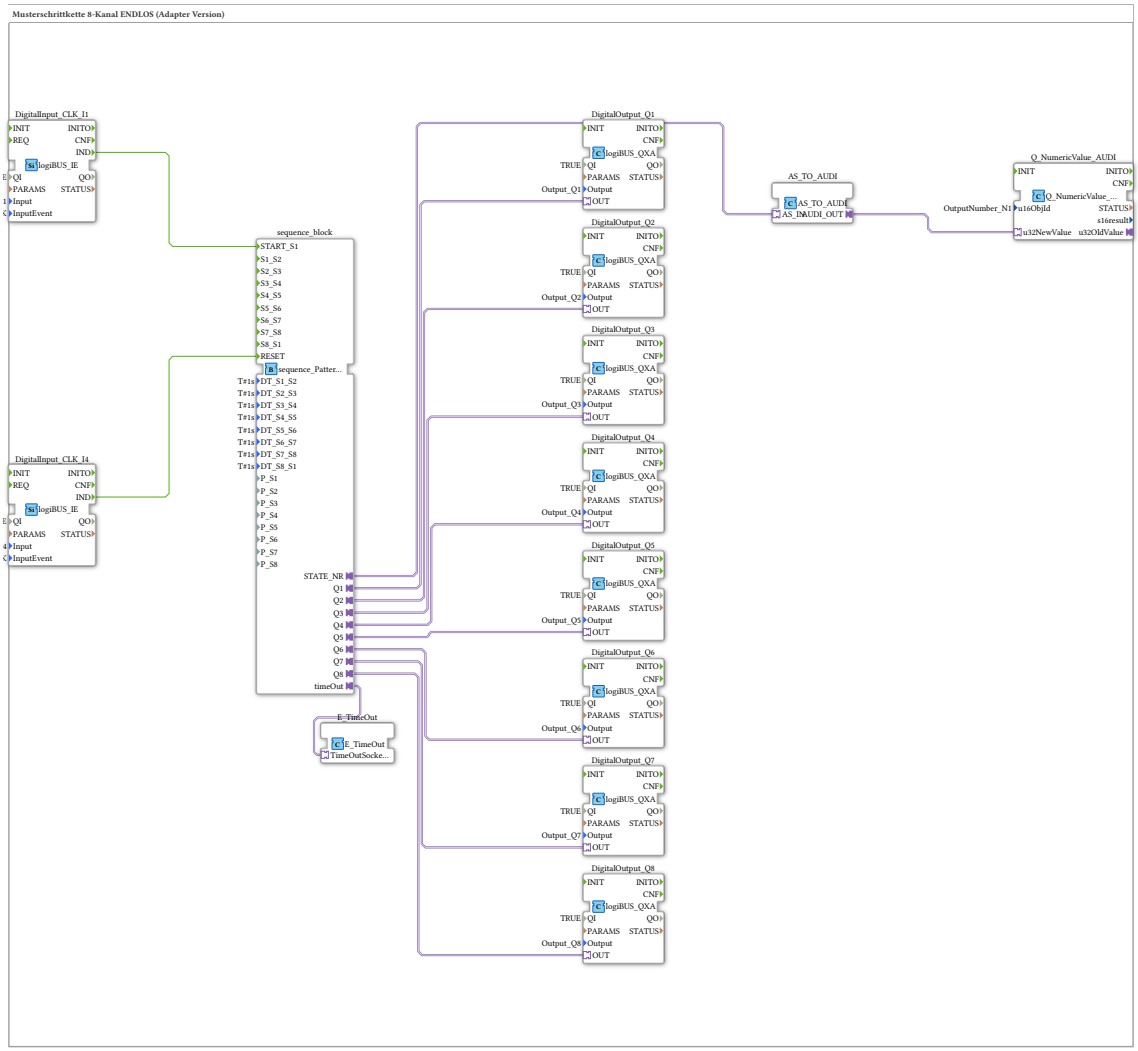
Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_064_AX`. Wir bauen eine musterbasierte Lauflicht-Sequenz mit AX-Adaptertechnologie.

Ziel der Übung

Realisierung einer Lauflicht-Sequenz über 8 Ausgänge (`Q1-Q8`), die im Gegensatz zu den einfachen Schrittketten (`Uebung_044_AX` bis `Uebung_063_AX`) auf einem musterbasierten Sequenzer beruht.

Beschreibung und Komponenten

Die Subapplikation `Uebung_064_AX` nutzt den Baustein `sequence_Pattern_08_08_loop_AX`. Anders als die einfachen Zustands-Sequenzer (`sequence_T_08_AX`, `sequence_E_08_AX`) adressiert dieser Baustein seine Ausgänge direkt als `Q1-Q8` statt über einzelne `DO_Sx`-Zustandsausgänge - er kann damit beliebige Bitmuster (nicht nur „ein Kanal nach dem anderen“) über die Zeit ausgeben.



- **`sequence_Pattern_08_08_loop_AX`**: Musterbasierter 8-Kanal-Sequenzer, zeitgesteuert über dieselben `DT_Sx_Sy`-Parameter wie die einfachen Zeitsteuerungs-Übungen, gibt aber je Schritt ein komplettes 8-Bit-Muster aus.
- **`E_TimeOut`**: Überwacht die Sequenz als Watchdog.
- **`Q_NumericValue_AUDI`**: Zeigt die aktuelle Schrittnummer auf dem ISOBUS-Terminal an.

Funktionsweise

1. Start durch Taster `I1` -> Sequenz startet.
2. Nach jeweils `T#1s` wechselt das ausgegebene Bitmuster zum nächsten Schritt.
3. Nach dem letzten Schritt (`S8`) kehrt die Sequenz automatisch zu `S1` zurück (ENDLOS).
4. Reset durch Taster `I4` -> Alles aus.

2.1.2.342 Übung 064: Musterschrittkette 8-Kanal ENDLOS

Thema: Ablaufsteuerungen in der Landtechnik

Situationsbeschreibung

Für ein System in der landwirtschaftlichen Automatisierung (z.B. eine Fassreinigung, eine Klappvorrichtung oder ein Gebläse) soll eine Ablaufsteuerung implementiert werden. Es werden 8 Stufen oder Aktoren nacheinander geschaltet.

Die Steuerung soll über den Baustein `sequence_Pattern_08_08_loop_AX` realisiert werden.

Funktionsbeschreibung der Ablaufsteuerung

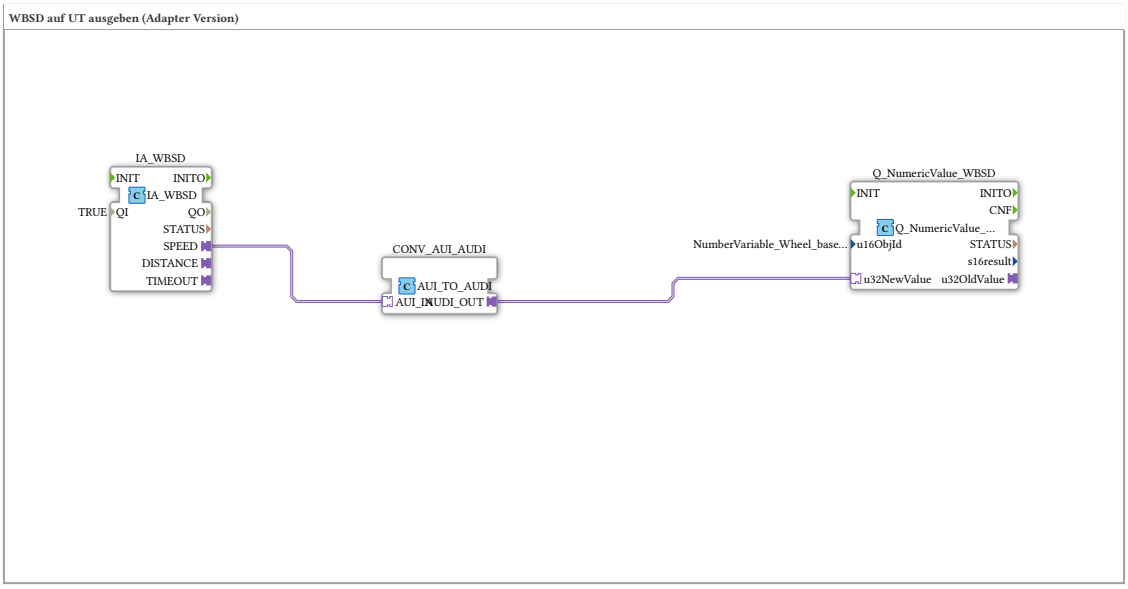
- **Start:** Durch Betätigen des Tasters `I1` (`START_S1`).

- **Phasen:** 8 aufeinanderfolgende Ausgangsschritte (Ausgänge `Q1` bis `Q8`).
- **Übergänge:** Die Schrittübergänge schalten Bit-Muster (Nockenwalze) zeitgesteuert durch.
- **Reset:** Zu jedem Zeitpunkt kann die Schrittkette über einen Reset (z.B. Taster `I4` oder Softkey) in den Ausgangszustand (Schritt 0) zurückgesetzt werden.

Arbeitsauftrag

1. Öffnen Sie das 4diac-Projekt und legen Sie die SubApp `Uebung_064_AX` bzw. `Uebung_064` an.
2. Platzieren Sie den Sequenzer-Baustein `sequence_Pattern_08_08_loop_AX` und konfigurieren Sie die Parameter.
3. Verbinden Sie den Start-Taster `I1` und den Reset-Taster `I4` mit den entsprechenden Eingängen.
4. Schalten Sie die 8 Ausgänge auf die physischen Ausgänge `Output_Q1` bis `Output_Q8`.
5. Zur Visualisierung soll die Zustandsnummer `STATE_NR` über den Baustein `AS_TO_AUDI` (Konvertierung `AS` ⟶ `AUDI`) auf dem ISOBUS Virtual Terminal-Ausgang `OutputNumber_N1` angezeigt werden.

2.1.2.343 Uebung_070_AUI: WBSD auf UT ausgeben (Adapter Version)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert die Ausgabe der radbasierten Maschinengeschwindigkeit (Wheel-Based Machine Speed – WBSD) auf einem Universal Terminal (UT) unter Verwendung von Adaptern. Im Gegensatz zur Basisübung (Uebung_070) kommt hier eine Adapter-basierte Verbindung zwischen dem Sensor-Interface und dem Ausgabebaustein zum Einsatz. Die Kommunikation erfolgt über das proprietäre AUI-Protokoll, welches für die Anbindung an den UT-Baustein in ein AUDI-Protokoll konvertiert werden muss.

Verwendete Funktionsbausteine (FBs)

Die Übung besteht aus drei vordefinierten Funktionsbausteinen, die im SubApp-Netzwerk miteinander verbunden sind:

- **IA_WBSD** – Eingangsadapter für die radbasierte Maschinengeschwindigkeit
- **Q_NumericValue_WBSD** – Ausgabebaustein zur Darstellung des numerischen Werts auf dem UT
- **CONV_AUI_AUDI** – Adapter-Konvertierer von AUI nach AUDI

IA_WBSD

- **Typ**: `isobus::tecu::IA_WBSD`
- **Parameter**:
  - `QI` = `TRUE` (Baustein ist aktiv)

- **Funktionsweise**: Der Baustein stellt die aktuellen Geschwindigkeitsdaten (wheel-based machine speed) über einen **Adapter-Ausgang** (`SPEED`) zur Verfügung. Er dient als Schnittstelle zur Fahrzeug-Sensorik.

Q_NumericValue_WBSD

- **Typ**: `isobus::UT::Q::Q_NumericValue_AUDI`
- **Parameter**:
  - `u16ObjId` = `NumberVariable_Wheel_based_machine_speed` (Verweis auf die Objekt-ID des numerischen Variablen-Eintrags im UT)
- **Funktionsweise**: Über diesen Baustein wird ein numerischer Wert an das Universal Terminal gesendet. Die Objekt-ID legt fest, welche Variable (hier die Radgeschwindigkeit) auf dem UT visualisiert wird.

CONV_AUI_AUDI

- **Typ**: `adapter::conversion::unidirectional::AUI_TO_AUDI`
- **Parameter**: keine
- **Funktionsweise**: Dieser Baustein konvertiert die AUI-Adapter-Schnittstelle (`AUI_IN`) in eine AUDI-Adapter-Schnittstelle (`AUDI_OUT`). Er ermöglicht so die Kommunikation zwischen dem IA_WBSD (AUI-basiert) und dem Q_NumericValue_WBSD (AUDI-basiert).

Programmablauf und Verbindungen

Der Datenfluss in der Übung ist wie folgt:

1. Der Baustein **IA_WBSD** erfasst die radbasierte Maschinengeschwindigkeit und stellt sie über den Adapter-Ausgang `SPEED` bereit.
2. Die Verbindung `IA_WBSD.SPEED → CONV_AUI_AUDI.AUI_IN` überträgt das AUI-Signal an den Konvertierungsbaustein.
3. **CONV_AUI_AUDI** wandelt das AUI-Protokoll in das AUDI-Protokoll um und gibt das Ergebnis am Ausgang `AUDI_OUT` aus.
4. Die Verbindung `CONV_AUI_AUDI.AUDI_OUT → Q_NumericValue_WBSD.u32NewValue` liefert den konvertierten Datenwert an den Ausgabebaustein.
5. **Q_NumericValue_WBSD** sendet den Wert unter der zugewiesenen Objekt-ID (`NumberVariable_Wheel_based_machine_speed`) an das UT, wo die Geschwindigkeit angezeigt wird.

Die gesamte Kommunikation erfolgt rein über Adapterverbindungen – es werden weder Ereignis- noch Datenleitungen im herkömmlichen Sinne verwendet.

Zusammenfassung

Die Übung **Uebung_070_AUI** vermittelt folgende Lerninhalte:

- Nutzung von **Adapter-Verbindungen** in IEC 61499-Anwendungen
- **Protokollkonvertierung** zwischen AUI und AUDI mittels spezialisierter Konvertierungsbausteine
- **Ausgabe numerischer Variablen** auf einem Universal Terminal über vordefinierte Objekt-IDs
- Verständnis des **Datenflusses** zwischen Sensor-Interface (IA) und UT-Ausgabe (Q)

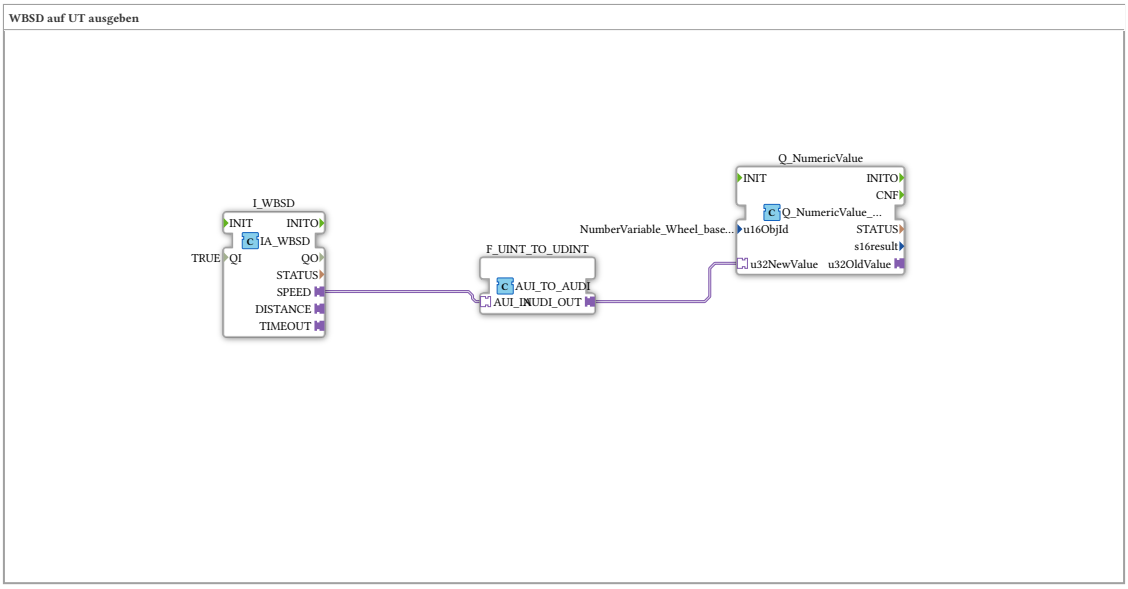
Der Schwierigkeitsgrad ist als **fortgeschritten** einzustufen, da grundlegende Kenntnisse über Adapter-Schnittstellen und das UT-System vorausgesetzt werden. Die Übung kann direkt in der 4diac-IDE geladen und simuliert werden.

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

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

2.1.2.344 Uebung_070_AX: WBSD auf UT ausgeben

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_070_AX (WBSD auf UT ausgeben).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **WBSD auf UT ausgeben**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_070_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **I_WBSD**: Instanz des Typs isobus::tecu::IA_WBSD.
  - Parameter QI = TRUE
- **Q_NumericValue**: Instanz des Typs isobus::UT::Q::Q_NumericValue_AUDI.
  - Parameter u16ObjId = NumberVariable_Wheel_based_machine_speed
- **F_UINT_TO_UDINT**: Instanz des Typs adapter::conversion::unidirectional::AUI_TO_AUDI.

Verbindungen und Schnittstellen

Adapterverbindungen:

- I_WBSD.SPEED -> F_UINT_TO_UDINT.AUI_IN
- F_UINT_TO_UDINT.AUDI_OUT -> Q_NumericValue.u32NewValue

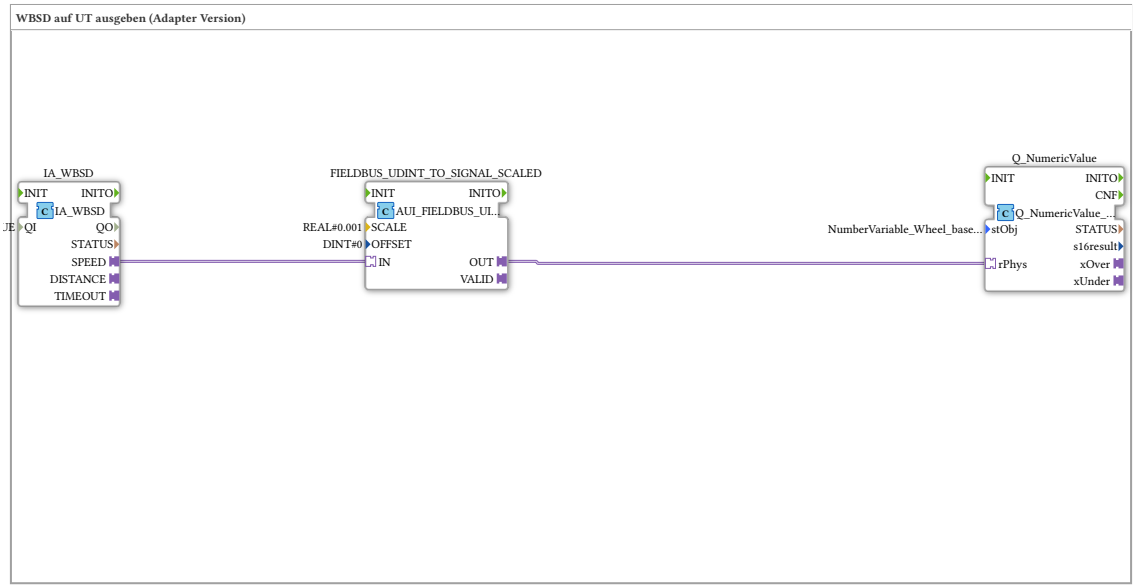
Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_070_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.345 Uebung_070c_AUI: WBSD auf UT ausgeben (Adapter Version)



Kein Bild vorhanden.

• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung zeigt, wie eine radbasierte Maschinengeschwindigkeit (WBSD – Wheel Based Speed) über den ISOBUS erfasst und auf ein UT (Universal Terminal) ausgegeben wird. Die Implementierung erfolgt als Subapplikation (SubApp) und nutzt Adapterverbindungen zur Kommunikation zwischen den Funktionsbausteinen. Ziel ist es, den Eingangswert mittels eines Adapters zu skalieren und als numerischen Wert auf das UT zu bringen.

Verwendete Funktionsbausteine (FBs)

Die SubApp enthält drei Funktionsbausteine, die über Adapter verbunden sind. Es gibt keine weiteren Sub-Bausteine (SubApp-Typ ist selbst bereits die oberste Ebene).

IA_WBSD

- **Typ**: `isobus::tecu::IA_WBSD`
- **Beschreibung**: Dieser Funktionsbaustein stellt den Adapter für die radbasierte Geschwindigkeit (WBSD) bereit. Er liefert einen Geschwindigkeitswert über den Ausgangsadapter `SPEED`.
- **Parameter**:
  - `QI` = `TRUE` (Qualifier für die Aktivierung)

FIELDBUS_UDINT_TO_SIGNAL_SCALED

- **Typ**: `logiBUS::signalprocessing::fieldbus::AUI_FIELDBUS_UINT_TO_SIGNAL_SCALED`
- **Beschreibung**: Dieser Baustein skaliert den eingehenden Ganzzahlwert (UDINT) in einen physikalischen Signalwert. Er wird verwendet, um den von `IA_WBSD` gelieferten Geschwindigkeitswert in eine für das UT verständliche Einheit umzurechnen.
- **Parameter**:
  - `SCALE` = `REAL#0.001` (Skalierungsfaktor)
  - `OFFSET` = `DINT#0` (Offset)

Q_NumericValue

- **Typ**: `isobus::UT::Q::Q_NumericValue_PHYSA`
- **Beschreibung**: Dieser Funktionsbaustein stellt einen numerischen Wert (z. B. die Geschwindigkeit) auf dem Universal Terminal dar. Er referenziert eine UT-Variable aus dem ISOBUS-Pool.
- **Parameter**:
  - `stObj` = `NumberVariable_Wheel_based_machine_speed` (Referenz auf die entsprechende Variable im UT-Pool; importiert aus `Uebungen::const::UT::TECU::DefaultPool_TECU_Numeric`)

Programmablauf und Verbindungen

Die SubApp besitzt keine eigenen Ein-/Ausgangsschnittstellen (SubAppInterfaceList ist leer). Die gesamte Datenverarbeitung erfolgt intern über Adapterverbindungen:

1. Der Block `IA_WBSD` empfängt die radbasierte Geschwindigkeit (vermutlich vom ISOBUS-Netzwerk) und gibt sie über seinen Adapterausgang `SPEED` aus.
2. Über eine Adapterverbindung wird `SPEED` an den Eingang `IN` des Skalierungsbausteins `FIELDBUS_UDINT_TO_SIGNAL_SCALED` weitergeleitet.
3. Der Skalierungsbaustein multipliziert den Wert mit `0.001` und addiert kein Offset (0). Das Ergebnis steht am Ausgang `OUT` bereit.
4. Über eine weitere Adapterverbindung gelangt das skalierte Signal an den `rPhys`-Eingang des Bausteins `Q_NumericValue`.
5. Der Baustein `Q_NumericValue` setzt den empfangenen physikalischen Wert auf die UT-Variable `NumberVariable_Wheel_based_machine_speed`, sodass die Geschwindigkeit am Terminal angezeigt wird.

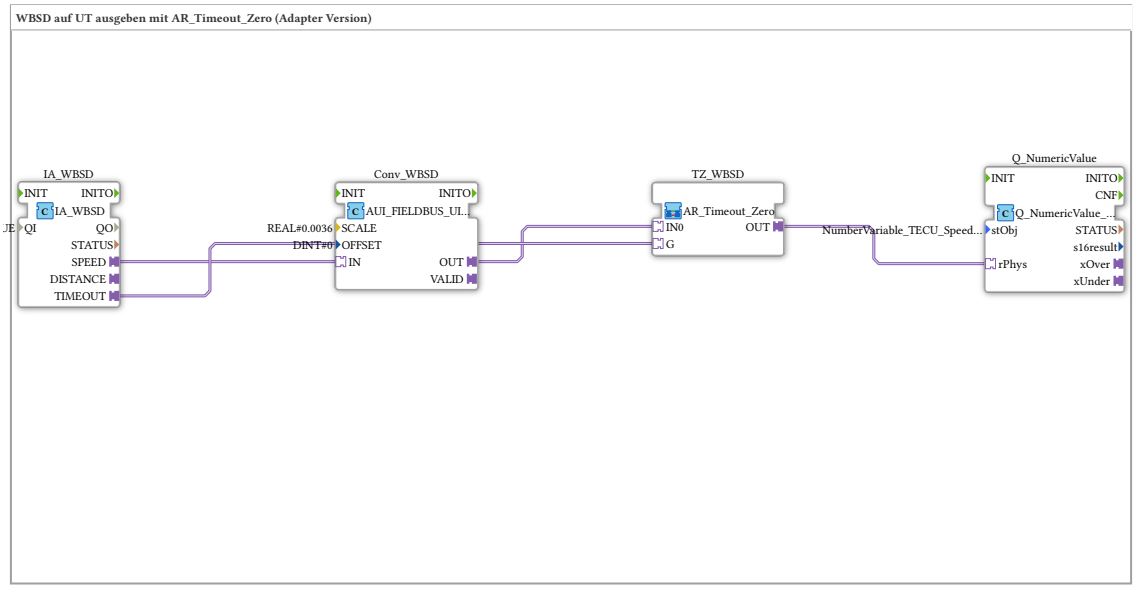
Besonderheiten:

- Die Übung arbeitet **ohne externe Events** – die Blöcke sind rein datengetrieben (keine sichtbaren Ereignisverbindungen).
- Die verwendeten Adapter erlauben eine flexible Kopplung der Funktionen, ohne dass alle Bausteine im selben Netzwerk liegen müssen.
- Die UT-Variable muss im Zielsystem als `NumberVariable_Wheel_based_machine_speed` vorhanden sein (Import aus dem Pool `DefaultPool_TECU_Numeric`).

Zusammenfassung

Die Übung `Uebung_070c_AUI` demonstriert die skalierte Ausgabe einer radbasierten Maschinengeschwindigkeit auf ein Universal Terminal mittels Adaptertechnik. Der Wert wird zunächst über den ISOBUS-Adapter erfasst, mit einem Faktor von 0.001 skaliert und anschließend als numerische Variable auf dem UT dargestellt. Dieses Vorgehen ist typisch für die Anbindung von Sensordaten an eine Bedienoberfläche in landwirtschaftlichen Maschinen nach ISOBUS-Standard.

2.1.2.346 Uebung_070d_AUI: WBSD auf UT ausgeben mit AR_Timeout_Zero (Adapter-Version)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung gibt die radbasierte Geschwindigkeit (Wheel Based Speed and Distance, WBSD) des TECU auf dem ISOBUS-Terminal aus: mit PHYS-Skalierung und der Fendt-Schaltung `AR_Timeout_Zero` (Adap-

ter-Version, analog zur Krauternter-SubApp `TECU_WBSD.SUB`). Bei einem Kommunikationsausfall wird `0.0` angezeigt, statt den letzten Wert einzufrieren.

Verwendete Funktionsbausteine (FBs)

- **IA_WBSD**: ISOBUS-Adapter für Wheel Based Speed and Distance (WBSD)
  - **Typ**: `isobus::tecu::IA_WBSD`
  - **Parameter**: `QI = TRUE`
  - **Erklärung**: Liefert die radbasierte Geschwindigkeit als `AUI`-Adapter `SPEED` und den Timeout-Status als `AX`-Adapter `TIMEOUT`.
- **Conv_WBSD**: Skalierung des Feldbuswerts
  - **Typ**: `logiBUS::signalprocessing::fieldbus::AUI_FIELDBUS_UINT_TO_SIGNAL_SCALED`
  - **Parameter**: `SCALE = 0.0036`, `OFFSET = 0`
  - **Erklärung**: Wandelt den `AUI`-Wert (UINT vom Feldbus) nur bei gültigem Signal in einen skalierten `AR`-Wert (REAL) um; `OFFSET` wird nach der Skalierung addiert.
- **TZ_WBSD**: Fendt-Schaltung (SubApp)
  - **Typ**: `MyLib::sys::AR_Timeout_Zero`
  - **Parameter**: keine
  - **Erklärung**: Gibt `IN0` unverändert aus, außer `G` (hier `IA_WBSD.TIMEOUT`, echter Kommunikationsausfall) ist `TRUE`: Dann wird `0.0` statt des sonst eingefrorenen letzten Werts ausgegeben.
- **Q_NumericValue**: Anzeige auf dem ISOBUS-Terminal
  - **Typ**: `isobus::UT::Q::Q_NumericValue_PHYSA`
  - **Parameter**: `stObj = NumberVariable_TECU_Speed_WBSD_N`
  - **Erklärung**: Zeigt den physikalischen Wert (`rPhys`) im Objekt `NumberVariable_TECU_Speed_WBSD_N` an.

Programmablauf und Verbindungen

1. `IA_WBSD.SPEED` ⟶ `Conv_WBSD.IN` (AdapterConnection): Der Geschwindigkeitswert vom TECU wird skaliert.
2. `Conv_WBSD.OUT` ⟶ `TZ_WBSD.IN0` (AdapterConnection): Der skalierte Wert ist der Live-Wert der Fendt-Schaltung.
3. `IA_WBSD.TIMEOUT` ⟶ `TZ_WBSD.G` (AdapterConnection): Der Timeout-Status steuert, ob `0.0` ausgegeben wird.
4. `TZ_WBSD.OUT` ⟶ `Q_NumericValue.rPhys` (AdapterConnection): Das Ergebnis wird auf dem Terminal angezeigt.

Verhalten: Solange der TECU die Botschaft liefert, zeigt das Terminal den mit `0.0036` skalierten Wert. Fällt die Kommunikation aus (`TIMEOUT`), zeigt es `0.0`.

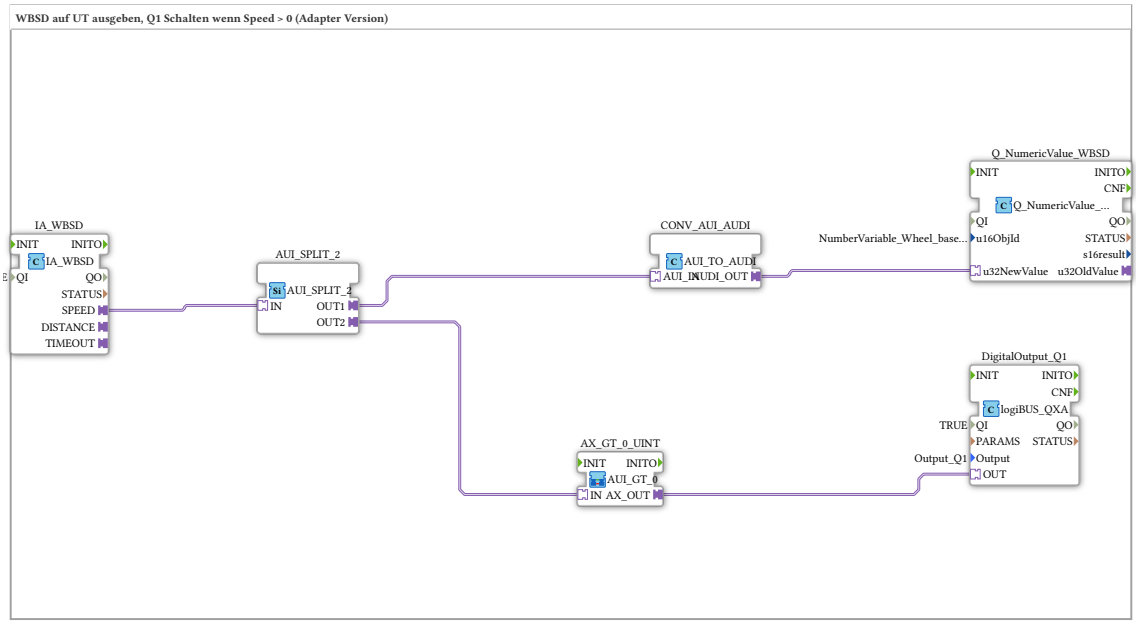
Zusammenfassung

Die Übung zeigt die Kette TECU-Adapter ⟶ Skalierung ⟶ Fendt-Schaltung ⟶ Anzeige. Die wiederverwendbare SubApp `AR_Timeout_Zero` (aus `MyLib_AX`) kapselt dabei die Auswahl zwischen Live-Wert und `0.0`.

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

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

2.1.2.347 Uebung_071_AUI: WBSD auf UT ausgeben, Q1 Schalten wenn Speed > 0 (Adapter Version)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert die Verwendung von Adaptern und einer benutzerdefinierten SubApp, um die **Wheel Based Machine Speed (WBSD)** von einem ISOBUS-IA-WBSD-Baustein über den **Universal Task (UT)** auszugeben und gleichzeitig einen digitalen Ausgang Q1 zu schalten, sobald die Geschwindigkeit größer als 0 ist. Die gesamte Kommunikation erfolgt über Adapter-Schnittstellen, was eine modulare und typsichere Verbindung der Funktionsbausteine ermöglicht.

Verwendete Funktionsbausteine (FBs)

Haupt-FBs (auf oberster Ebene)

- **IA_WBSD**: `isobus::tecu::IA_WBSD` ISOBUS-Adapter-Baustein für die Wheel Based Machine Speed. Parameter: `QI` = TRUE (aktiviert).
- **Q_NumericValue_WBSD**: `isobus::UT::Q::Q_NumericValue_AUDI` Baustein zum Senden eines numerischen Werts (Speed) an den UT. Parameter: `u16ObjId` = `NumberVariable_Wheel_based_machine_speed` (Objektreferenz aus konstantem Pool).
- **DigitalOutput_Q1**: `logiBUS::io::DQ::logiBUS_QXA` Digitalausgangsbaustein für den logiBUS. Parameter: `QI` = TRUE, `Output` = `Output_Q1` (definierte Konstante für den Ausgang).
- **CONV_AUI_AUDI**: `adapter::conversion::unidirectional::AUI_TO_AUDI` Konvertiert einen AUI-Adapter (unidirektional) in einen AUDI-Adapter (unidirektional) – vermutlich zur Anpassung der Schnittstelle.
- **AUI_SPLIT_2**: `adapter::events::unidirectional::AUI_SPLIT_2` Verteilt ein eingehendes AUI-Ereignis auf zwei Ausgänge (OUT1, OUT2) – hier für parallele Weiterleitung der Geschwindigkeitsdaten.

Sub-Bausteine: `AX_GT_0_UINT`

- **Typ**: `MyLib::sys::AUI_GT_0` (benutzerdefinierte SubApp)
- **Verwendete interne FBs**: (Details liegen nicht vor, da die SubApp extern definiert ist. Sie enthält vermutlich einen Vergleichsbaustein für unsigned integer.)
- **Funktionsweise**: Diese SubApp prüft, ob der eingehende Wert (vom Typ UINT) größer als 0 ist. Trifft dies zu, wird der Ausgangsadapter `AX_OUT` aktiviert. Dieser Ausgang steuert anschließend den digitalen Ausgang Q1 (über den Adapterverbund mit `DigitalOutput_Q1.OUT`).

Programmablauf und Verbindungen

1. Der Adapter `IA_WBSD` liefert kontinuierlich die aktuelle Radgeschwindigkeit über den Adapterausgang `SPEED` (AUI-Format).
2. Der Split-Baustein `AUI_SPLIT_2` empfängt die Geschwindigkeit und leitet sie an zwei Pfade weiter:
   - **OUT1** → `CONV_AUI_AUDI` → `Q_NumericValue_WBSD`: Die Geschwindigkeit wird über den UT als numerischer Wert ausgegeben (Objektreferenz `NumberVariable_Wheel_based_machine_speed`).
   - **OUT2** → `AX_GT_0_UINT`: Die Geschwindigkeit wird auf > 0 geprüft.
3. Der SubApp `AX_GT_0_UINT` aktiviert bei positiver Prüfung den Ausgangsadapter `AX_OUT`.
4. Der Adapterausgang `AX_OUT` speist den Eingang `OUT` des Digitalausgangsbausteins `DigitalOutput_Q1`, sodass Q1 (z. B. ein Relais oder eine Lampe) eingeschaltet wird, solange die Geschwindigkeit größer als 0 ist.

Lernziele:

- Verwendung von ISOBUS- und logiBUS-Bausteinen in 4diac.
- Arbeiten mit Adaptern (AUI/AUDI) und Adapter-Splittern.
- Einbindung einer selbst erstellten SubApp (AX_GT_0_UINT) in ein größeres Netzwerk.
- Praxisnahe Automatisierungsaufgabe: Geschwindigkeitsausgabe und Schwellwertüberwachung.

Schwierigkeitsgrad: Fortgeschritten (Grundkenntnisse in IEC 61499 und Adapterkonzepten erforderlich).

Hinweis: Die Übung liegt als **Adapter-Version** der Grundübung `Uebung_071` vor. Voraussetzung ist ein funktionierender ISOBUS-UT-sowie logiBUS-Ausgang in der Zielumgebung.

Zusammenfassung

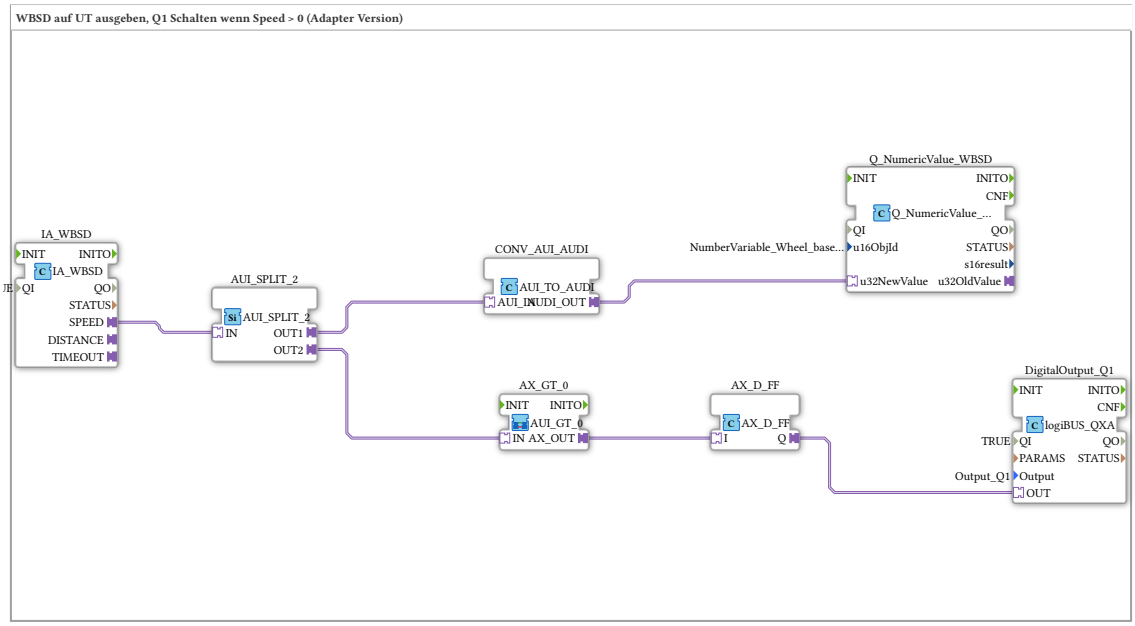
Die Übung `Uebung_071_AUI` zeigt eine typische landwirtschaftliche Automatisierungsaufgabe: die Ausgabe einer Maschinengeschwindigkeit auf einem Universal Terminal und die gleichzeitige Aktivierung eines digitalen Ausgangs bei Bewegung. Alle Kommunikationsverbindungen wurden durch Adaptertypen realisiert, was die Wiederverwendbarkeit und Austauschbarkeit der Bausteine erhöht. Der Einsatz der SubApp `AX_GT_0_UINT` verdeutlicht, wie eigene kleine Logikbausteine dezentral in das Gesamtsystem integriert werden.

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

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

2.1.2.348 Uebung_071a_AUI: WBSD auf UT ausgeben, Q1 Schalten wenn Speed > 0 (Adapter Version)

(WBSD auf UT ausgeben, Q1 Schalten wenn Speed > 0 – Adapter Version)



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung demonstriert die Verwendung von Adapter-Interfaces (AUI/AUDI) in 4diac, um eine radbasierte Maschinengeschwindigkeit (WBSD) auszulesen, auf einem Universal Terminal (UT) darzustellen und einen digitalen Ausgang (Q1) zu schalten, sobald die Geschwindigkeit größer als Null ist. Die gesamte Logik ist als SubApp mit Adapterverbindungen realisiert.

Verwendete Funktionsbausteine (FBs)

Sub-Baustein: `AX_GT_0`

- **Typ**: `MyLib::sys::AUI_GT_0` (SubApp)
- **Verwendete interne FBs**: Nicht näher spezifiziert (gehört zur Bibliothek `MyLib::sys`)
- **Funktionsweise**: Dieser Baustein nimmt über einen Adapter-Eingang (AUI) einen ganzzahligen Wert entgegen und prüft, ob dieser größer als Null ist. Am Ausgang `AX_OUT` wird ein boolesches Signal (TRUE/FALSE) bereitgestellt, das das Ergebnis des Vergleichs darstellt.

Weitere verwendete Funktionsbausteine

Bausteinname	Typ	Parameter	Kurzbeschreibung
`IA_WBSD`	`isobus::tecu::IA_WBSD`	`QI` = TRUE	Liefert die radbasierte Maschinengeschwindigkeit über ein AUI-Interface.
`AUI_SPLIT_2`	`Adapter::events::unidirectional::AUI_SPLIT_2`	–	Verteilt ein eingehendes AUI-Signal auf zwei identische Ausgänge.
`CONV_AUI_AUDI`	`Adapter::conversion::unidirectional::AUI_TO_AUDI`	–	Konvertiert ein AUI-Signal in das AUDI-Format, das von UT-Anzeigebausteinen erwartet wird.
`Q_NumericValue_WBSD`	`isobus::UT::Q::Q_NumericValue_Variable_Wheel_based_machine_speed`	`Q_ObjectID` = Wert aus Pool	Zeigt den numerischen Wert der Geschwindigkeit auf dem UT an (Object-ID aus der Pool-Konfiguration).
`AX_GT_0`	`MyLib::sys::AUI_GT_0`	– (SubApp)	Prüft, ob der eingehende Wert > 0 ist (siehe oben).
`AX_D_FF`	`Adapter::events::unidirectional::AX_D_FF`	–	Setzt ein Flip-Flop, das den booleschen Zustand hält und an den Ausgang weitergibt.
`DigitalOutput_Q1`	`logiBUS::io::DQ::logiBUS_QXX`	`QI` = TRUE, `Output` = `Output_Q1`	Schaltet den digitalen Ausgang Q1 des logiBUS-Moduls entsprechend des anliegenden Signals.

Programmablauf und Verbindungen

1. Der Baustein `IA_WBSD` liefert die aktuelle Maschinengeschwindigkeit als AUI-Datensignal.
2. Dieses Signal wird über eine Adapterverbindung an `AUI_SPLIT_2` weitergeleitet, das es auf zwei parallele Pfade aufteilt:

- **Pfad 1 (Anzeige)**: Über `CONV_AUI_AUDI` wird das AUI-Signal in ein AUDI-Signal umgewandelt. Dieses wird an `Q_NumericValue_WBSD` übergeben, das den Zahlenwert auf dem Universal Terminal (UT) anzeigt. Die Object-ID `NumberVariable_Wheel_based_machine_speed` legt fest, welcher Parameter der Pool-Konfiguration dargestellt wird.
- **Pfad 2 (Schwellwertprüfung)**: Das AUI-Signal wird direkt an den Sub-Baustein `AX_GT_0` angeschlossen. Dieser prüft, ob der Wert größer als Null ist.

3. Das Ergebnis der Prüfung (`AX_OUT`) wird an den Flip-Flop-Baustein `AX_D_FF` übergeben. Dieser stabilisiert das Signal und verhindert kurzzeitiges Flattern.
4. Der Ausgang von `AX_D_FF` wird über eine Adapterverbindung an `DigitalOutput_Q1` geführt. Wenn die Geschwindigkeit > 0 ist, schaltet `DigitalOutput_Q1` den logiBUS-Ausgang Q1 ein; andernfalls bleibt Q1 aus.

Abhängigkeiten:

- Die Konstanten `NumberVariable_Wheel_based_machine_speed` und `Output_Q1` müssen im Projekt als `Uebungen::const::UT::TECU::DefaultPool_TECU` bzw. `logiBUS::io::DQ::logiBUS_DO` definiert sein.
- Der Baustein `logiBUS_QXA` benötigt eine gültige logiBUS-Hardwarekonfiguration.

Zusammenfassung

Die Übung **Uebung_071a_AUI** zeigt eine typische Anwendung von Adapter-Interfaces in der Automatisierungstechnik mit 4diac. Der Lernende wird mit folgenden Konzepten vertraut gemacht:

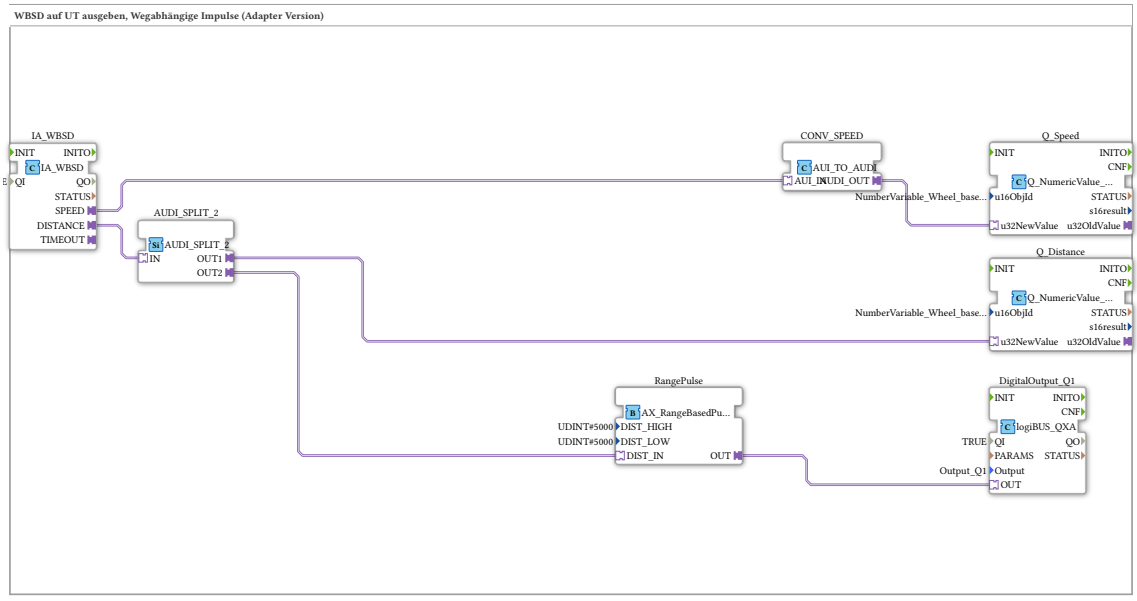
- **Adapter-Split, Konvertierung und Weiterleitung** (AUI, AUDI)
- **Auslesen eines ISOBUS-Sensors** (WBSD) und Darstellung auf einem UT
- **Schwellwertvergleich** und **Flip-Flop-Logik** für eine zuverlässige Ausgangsschaltung
- **Integration von logiBUS-Ausgängen** in ein Steuerungsprogramm

Nach erfolgreicher Bearbeitung versteht der Teilnehmer die Datenflussstruktur mit Adaptern und kann ähnliche Aufgaben selbstständig umsetzen.

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

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

2.1.2.349 Uebung_071b_AUI: WBSD auf UT ausgeben, Wegabhängige Impulse (Adapter Version)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert die Ausgabe von Wheel-Based Speed (WBSD) und Wheel-Based Distance (WBD) auf einem Universal Terminal (UT). Zusätzlich wird ein wegabhängiger Impuls erzeugt: Alle 10

Meter (5 Meter HIGH, 5 Meter LOW) schaltet ein digitaler Ausgang um. Die Realisierung erfolgt über Adapter-Schnittstellen und zeigt die typische Datenverarbeitungskette von der ISOBUS-Sensorerfassung bis zur UT-Anzeige und logiBUS-Ausgabe.

Verwendete Funktionsbausteine (FBs)

- **IA_WBSD** (`isobus::tecu::IA_WBSD`)
  - Parameter: `QI` = `TRUE`
  - Liest die fahrzeugseitige radbasierte Geschwindigkeit und Distanz (ISOBUS-TECU-Schnittstelle).
- **AUDI_SPLIT_2** (`adapter::events::unidirectional::AUDI_SPLIT_2`)
  - Verteilt ein eingehendes AUDI-Signal auf zwei identische Ausgänge (OUT1, OUT2).
- **CONV_SPEED** (`adapter::conversion::unidirectional::AUI_TO_AUDI`)
  - Wandelt das AUI-Signal von der WBSD-Schnittstelle in ein AUDI-Signal um, das von den UT- und logiBUS-Bausteinen verarbeitet werden kann.
- **Q_Speed** (`isobus::UT::Q::Q_NumericValue_AUDI`)
  - Parameter: `u16ObjId` = `NumberVariable_Wheel_based_machine_speed`
  - Zeigt die aktuelle Geschwindigkeit auf dem Universal Terminal an.
- **Q_Distance** (`isobus::UT::Q::Q_NumericValue_AUDI`)
  - Parameter: `u16ObjId` = `NumberVariable_Wheel_based_machine_distance`
  - Zeigt die aktuelle Distanz auf dem Universal Terminal an.
- **RangePulse** (`logiBUS::signalprocessing::distance::AX_RangeBasedPulse`)
  - Parameter: `DIST_HIGH` = `5000`, `DIST_LOW` = `5000`
  - Erzeugt ein pulsierendes Signal (HIGH/LOW) bei jeder Distanzänderung von insgesamt 10 Meter (5 Meter HIGH, 5 Meter LOW).
- **DigitalOutput_Q1** (`logiBUS::io::DQ::logiBUS_QXA`)
  - Parameter: `QI` = `TRUE`, `Output` = `Output_Q1`
  - Schaltet den logiBUS-Digitalausgang Q1 entsprechend dem empfangenen Signal.

Programmablauf und Verbindungen

1. **Geschwindigkeit** `IA_WBSD.SPEED` ⟶ `CONV_SPEED.AUI_IN` ⟶ `CONV_SPEED.AUDI_OUT` ⟶ `Q_Speed.u32NewValue` Die radbasierte Geschwindigkeit wird über einen Adapter-Konverter in ein AUDI-Signal umgewandelt und direkt auf dem UT (Numerische Anzeige) dargestellt.
2. **Distanz** `IA_WBSD.DISTANCE` ⟶ `AUDI_SPLIT_2.IN` Die Distanzinformation wird in zwei parallele Pfade aufgeteilt:
   - **Pfad 1 (UT-Anzeige)**: `AUDI_SPLIT_2.OUT1` ⟶ `Q_Distance.u32NewValue` Die Distanz wird ebenfalls auf dem UT angezeigt.
   - **Pfad 2 (Impulserzeugung)**: `AUDI_SPLIT_2.OUT2` ⟶ `RangePulse.DIST_IN` ⟶ `RangePulse.OUT` ⟶ `DigitalOutput_Q1.OUT` Der `RangePulse`-Baustein überwacht die Distanzänderung und erzeugt bei Erreichen von 5 m HIGH- und 5 m LOW-Schwellen einen Impuls. Dieser wird an den digitalen Ausgang Q1 weitergegeben, sodass Q1 periodisch mit 10 m-Zyklus schaltet.

Lernziele

- Adapter-basierte Signalverarbeitung zwischen ISOBUS, UT und logiBUS verstehen.
- Aufteilung eines Datensignals auf zwei getrennte Verarbeitungspfade anwenden.
- Wegabhängige Impulse mittels eines Distanz-Pulsgebers realisieren.

Schwierigkeitsgrad: Mittel **Voraussetzungen**: Grundkenntnisse in 4diac-IDE, ISOBUS- und logiBUS-Funktionsbausteinen. **Start**: Das SubApp-Template kann direkt in ein 4diac-Projekt importiert und mit einer gültigen Hardware-Konfiguration (z. B. TECU-Steuergerät) ausgeführt werden.

Zusammenfassung

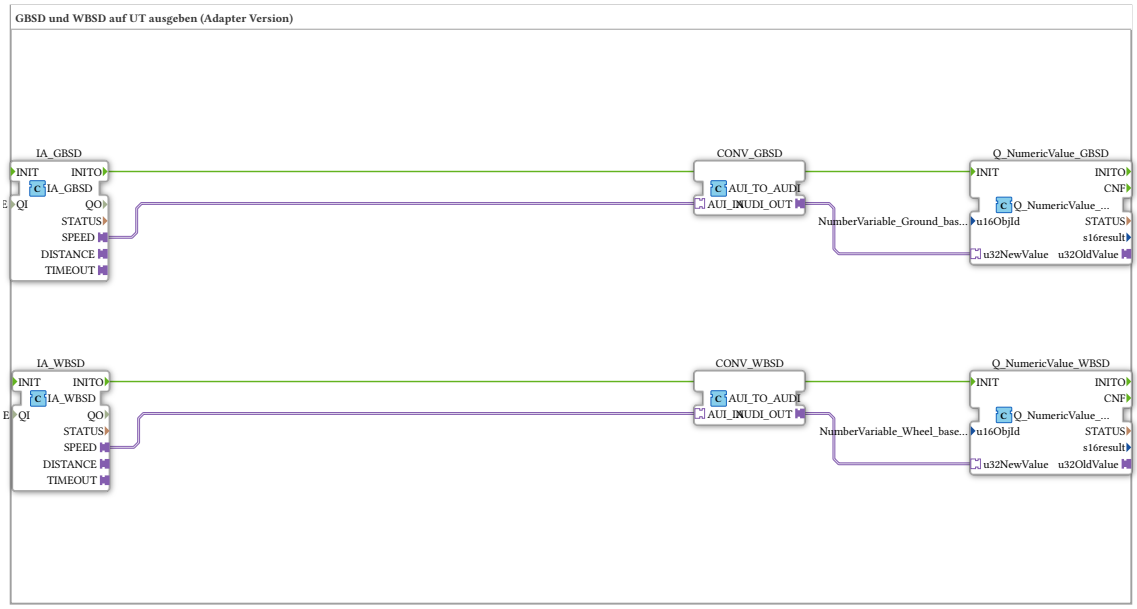
Die Übung zeigt eine vollständige Kette von der ISOBUS-Sensordatenaufnahme über Adapter-Konvertierung und Signalaufteilung bis hin zur UT-Anzeige und einer wegabhängigen digitalen Ausgabe. Die radbasierte Distanz wird genutzt, um alle 10 Meter einen Impuls auf einem logiBUS-Ausgang zu erzeugen.

Damit wird die typische Anwendung „WBSD auf UT ausgeben mit Wegimpulsen“ in einer adapter-basierten Variante umgesetzt.

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

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

2.1.2.350 Uebung_072_AUI: GBSD und WBSD auf UT ausgeben (Adapter Version)



• — ○ • — ○ • — ○ •

Einleitung

Diese Übung zeigt, wie die fahrzeugbasierte Geschwindigkeit (Ground-Based Machine Speed – GBSD) und die radbasierte Geschwindigkeit (Wheel-Based Machine Speed – WBSD) eines ISOBUS-TECU (Tractor Electronic Control Unit) über einen Interface-Adapter (AUI) ausgelesen und auf einem Universal Terminal (UT) dargestellt werden.
Die von den Adaptern bereitgestellten Geschwindigkeitsdaten werden zunächst mit einem unidirektionalen Konverter in ein numerisches AUDI-Interface umgewandelt und anschließend über entsprechende UT-Anzeigebausteine ausgegeben. Die dabei verwendeten Objekt-IDs stammen aus einem vordefinierten Konstantenpool.

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: `IA_GBSD`

- **Typ**: `isobus::tecu::IA_GBSD`
- **Parameter**:
  - `QI` = `TRUE`
- **Ereignisausgang**: `INITO`
- **Adapterausgang**: `SPEED` (AUI-Interface)
- **Funktionsweise**:
  Dieser Baustein liest die aktuelle fahrzeugbasierte Geschwindigkeit (GBSD) von der TECU aus und stellt sie über den Adapterausgang `SPEED` als AUI-Objekt bereit. Der Parameter `QI` muss auf `TRUE` gesetzt sein, um die Datenabfrage zu aktivieren.

Sub-Bausteine: `IA_WBSD`

- **Typ**: `isobus::tecu::IA_WBSD`
- **Parameter**:
  - `QI` = `TRUE`
- **Ereignisausgang**: `INITO`

- **Adapterausgang**: SPEED (AUI-Interface)
- **Funktionsweise**:
  Analog zu IA_GBSD stellt dieser Baustein die radbasierte Geschwindigkeit (WBSD) der TECU über den Adapterausgang SPEED als AUI-Objekt bereit.

Sub-Bausteine: CONV_GBSD

- **Typ**: adapter::conversion::unidirectional::AUI_TO_AUDI
- **Adaptereingang**: AUI_IN
- **Adapterausgang**: AUDI_OUT
- **Funktionsweise**:
  Der Konverter wandelt das eingehende AUI-Interface (z. B. ein Geschwindigkeitswert) in ein numerisches AUDI-Interface um. Dieses AUDI-Signal wird von den UT-Anzeigebausteinen als 32-Bit-Wert erwartet.

Sub-Bausteine: CONV_WBSD

- **Typ**: adapter::conversion::unidirectional::AUI_TO_AUDI
- **Adaptereingang**: AUI_IN
- **Adapterausgang**: AUDI_OUT
- **Funktionsweise**:
  Identisch zu CONV_GBSD. Wandelt das AUI-Signal der radbasierten Geschwindigkeit in ein AUDI-Signal.

Sub-Bausteine: Q_NumericValue_GBSD

- **Typ**: isobus::UT::Q::Q_NumericValue_AUDI
- **Parameter**:
  - u16ObjId = Uebungen::const::UT::TECU::DefaultPool_TECU::NumberVariable_Ground_based_machine_speed
- **Ereigniseingang**: INIT
- **Dateneingang**: u32NewValue (AUDI-Interface)
- **Funktionsweise**:
  Dieser Baustein zeigt einen numerischen Wert auf dem Universal Terminal an. Die Variable, die den Wert darstellt, wird durch die übergebene Objekt-ID (u16ObjId) bestimmt. Nach einem INIT-Ereignis wird der am Dateneingang anliegende Wert auf dem UT aktualisiert.

Sub-Bausteine: Q_NumericValue_WBSD

- **Typ**: isobus::UT::Q::Q_NumericValue_AUDI
- **Parameter**:
  - u16ObjId = Uebungen::const::UT::TECU::DefaultPool_TECU::NumberVariable_Wheel_based_machine_speed
- **Ereigniseingang**: INIT
- **Dateneingang**: u32NewValue (AUDI-Interface)
- **Funktionsweise**:
  Gleiche Funktionalität wie Q_NumericValue_GBSD, jedoch für die radbasierte Geschwindigkeit.

Programmablauf und Verbindungen

1. **Initialisierung**:
   Beide Interface-Adapter (IA_GBSD und IA_WBSD) werden mit QI = TRUE parametriert. Nach Systemstart generieren sie am Ausgang INIT0 ein Ereignis.
2. **Ereignisverkettung**:
   Das Ereignis INIT0 von IA_GBSD wird direkt mit dem Eingang INIT von Q_NumericValue_GBSD verbunden.
   Entsprechend wird INIT0 von IA_WBSD mit INIT von Q_NumericValue_WBSD verbunden. Dadurch werden die UT-Anzeigebausteine nach dem Auslesen der Daten initialisiert.
3. **Datenfluss (Adapter-Verbindungen)**:
   - Der Adapterausgang SPEED von IA_GBSD (AUI-Interface) wird mit dem Adaptereingang AUI_IN des Konverters CONV_GBSD verbunden.
   - Der Konverter CONV_GBSD wandelt das AUI-Interface in ein AUDI-Interface und gibt es an seinem Adapterausgang AUDI_OUT aus.
   - Dieser Ausgang wird mit dem Dateneingang u32NewValue von Q_NumericValue_GBSD verbunden.
   - Gleiches gilt für die radbasierte Geschwindigkeit (IA_WBSD ⟶ CONV_WBSD ⟶ Q_NumericValue_WBSD).

4. **Ergebnis**:
   Auf dem UT erscheinen zwei numerische Werte: die fahrzeugbasierte Geschwindigkeit (GBSD) und die radbasierte Geschwindigkeit (WBSD). Die Werte werden über die konfigurierten Objekt-IDs im Variablenpool des TECU bereitgestellt.

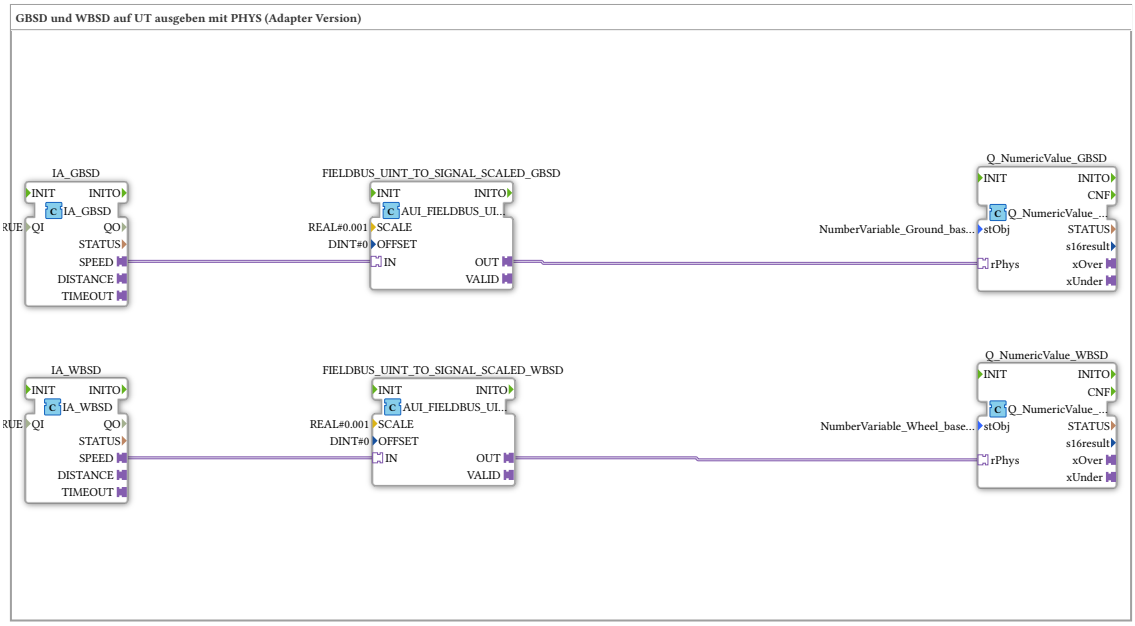
Lernziele:

- Verständnis des AUI- und AUDI-Interfacekonzepts in ISOBUS-Applikationen.
- Einsatz von unidirektionalen Adaptern zur Interface-Konvertierung.
- Verbindung von TECU-Ausgängen mit UT-Anzeigebausteinen.

Zusammenfassung

Die Übung demonstriert die vollständige Signalkette vom Auslesen zweier Geschwindigkeitsgrößen aus einem ISOBUS-TECU über eine adapterbasierte Konvertierung bis hin zur Anzeige auf einem Universal Terminal. Durch die klare Trennung von Adapter- und Daten-Interfaces wird die Wiederverwendbarkeit der Funktionsbausteine unterstützt und eine flexible Anbindung an unterschiedliche UT-Konfigurationen ermöglicht.

2.1.2.351 Uebung_072c_AUI: GBSD und WBSD auf UT ausgeben mit PHYS (Adapter Version)



• – ○ • – ○ • – ○ •

Einleitung

In dieser Übung wird das Verhalten der Funktionsbausteine `IA_GBSD` (Ground Based Machine Speed) und `IA_WBSD` (Wheel Based Machine Speed) auf einem ISOBUS Universal Terminal (UT) ausgegeben. Die von den jeweiligen ISOBUS-Applizierungen gelieferten digitalen Geschwindigkeitswerte (UINT) werden mittels eines Skalierungsfunktionsbausteins in physikalische Werte umgerechnet und anschließend über einen UT-Adapter (`Q_NumericValue_PHYSA`) auf dem Terminal dargestellt. Die Skalierung erfolgt mit einer Nachkommastellengenauigkeit von 0.001 (z. B. Umrechnung von mm/s in m/s).

Verwendete Funktionsbausteine (FBs)

- **IA_GBSD** (Typ: `isobus::tecu::IA_GBSD`)
  - Parameter: `QI` = TRUE
  - Liefert über den Adapterausgang `SPEED` einen UINT-Wert für die geschwindigkeitsbasierte Maschinengeschwindigkeit.
- **IA_WBSD** (Typ: `isobus::tecu::IA_WBSD`)
  - Parameter: `QI` = TRUE

  - Liefert über den Adapterausgang `SPEED` einen UINT-Wert für die radbasierte Maschinengeschwindigkeit.

- **FIELDBUS_UINT_TO_SIGNAL_SCALED_GBSD**
  (Typ: `logiBUS::signalprocessing::fieldbus::AUI_FIELDBUS_UINT_TO_SIGNAL_SCALED`)
  - Parameter:
    - `SCALE` = REAL#0.001
    - `OFFSET` = DINT#0
  - Skaliert den eingehenden UINT-Wert (IN) mit Faktor 0.001 und gibt das Ergebnis als REAL-Signal (OUT) weiter.
- **FIELDBUS_UINT_TO_SIGNAL_SCALED_WBSD**
  (Typ: `logiBUS::signalprocessing::fieldbus::AUI_FIELDBUS_UINT_TO_SIGNAL_SCALED`)
  - Parameter:
    - `SCALE` = REAL#0.001
    - `OFFSET` = DINT#0
  - Gleiche Funktionalität wie der Baustein für GBSD.
- **Q_NumericValue_GBSD** (Typ: `isobus::UT::Q::Q_NumericValue_PHYSA`)
  - Parameter: `stObj = NumberVariable_Ground_based_machine_speed` (importiert aus `Uebungen::const::UT::TECU::DefaultPool_TECU_Numeric`)
  - Stellt den physikalischen Wert (rPhys) auf dem UT dar.
- **Q_NumericValue_WBSD** (Typ: `isobus::UT::Q::Q_NumericValue_PHYSA`)
  - Parameter: `stObj = NumberVariable_Wheel_based_machine_speed` (importiert aus dem gleichen Pool)
  - Gleiche Funktionalität wie der Baustein für GBSD.

Programmablauf und Verbindungen

1. **Erfassung der Geschwindigkeiten**
   - Die Bausteine `IA_GBSD` und `IA_WBSD` werden mit aktivem `QI` betrieben und liefern kontinuierlich aktuelle Geschwindigkeitswerte als UINT-Daten an ihren Adapterausgängen `SPEED`.
2. **Skalierung**
   - Der Ausgang `SPEED` von `IA_GBSD` wird über eine Adapterverbindung an den Eingang `IN` von `FIELDBUS_UINT_TO_SIGNAL_SCALED_GBSD` angeschlossen.
   - Analog wird `SPEED` von `IA_WBSD` mit dem Eingang `IN` von `FIELDBUS_UINT_TO_SIGNAL_SCALED_WBSD` verbunden.
   - Beide Skalierungsbausteine multiplizieren den eingehenden UINT-Wert mit `0.001` (kein Offset) und geben das Ergebnis als REAL-Wert aus.
3. **Ausgabe auf dem UT**
   - Der skalierte Wert (Ausgang `OUT` des Skalierungsbausteins) wird als Datenquelle an den `rPhys`-Eingang des jeweiligen `Q_NumericValue_PHYSA`-Bausteins geführt.
   - Diese Bausteine sind mit den entsprechenden UT-Objekten (`NumberVariable_Ground_based_machine_speed` bzw. `NumberVariable_Wheel_based_machine_speed`) konfiguriert und stellen die Werte auf dem Universal Terminal dar.

Lernziele:

- Verständnis des Einsatzes von Adapterverbindungen in 4diac.
- Umsetzung einer physikalischen Skalierung von Feldbus-Rohdaten.
- Integration von ISOBUS-Applizierungsbausteinen mit UT-Ausgabebausteinen.

Schwierigkeitsgrad: Mittel **Vorkenntnisse:** Grundlagen der 4diac-IDE, ISOBUS-Terminal-Konfiguration, Arbeiten mit Adaptern.

Zusammenfassung

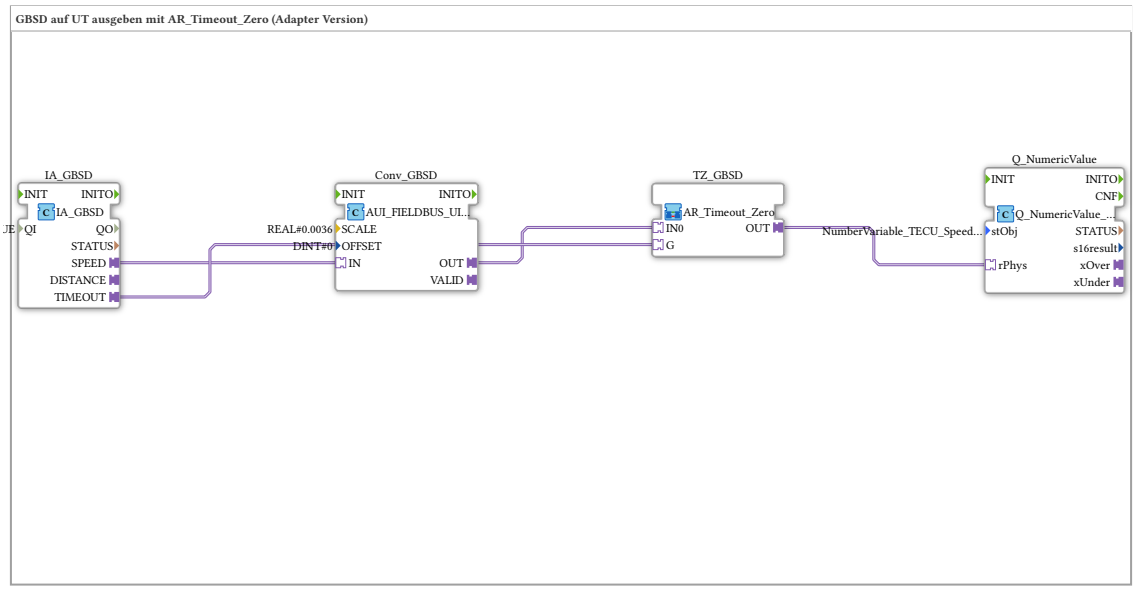
Die Übung demonstriert den durchgängigen Datenfluss von der ISOBUS-Applizierung (IA_GBSD / IA_WBSD) über eine lineare Skalierung (Faktor 0,001) bis zur visuellen Darstellung auf einem Universal

Terminal. Die Verwendung von Adaptern vereinfacht die Verbindung unterschiedlicher Bausteinschnittstellen und ermöglicht eine modulare Struktur. Als Ergebnis werden die aktuellen geschwindigkeitsbasierten und radbasierten Maschinengeschwindigkeiten physikalisch korrekt auf dem UT ausgegeben.

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

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

2.1.2.352 Uebung_072d_AUI: GBSD auf UT ausgeben mit AR_Timeout_Zero (Adapter-Version)



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung gibt die GBSD-Geschwindigkeit des TECU auf dem Universal Terminal (UT) aus. Der Rohwert wird mit `SCALE = 0.0036` in einen physikalischen `REAL`-Wert skaliert und durch die SubApp `AR_Timeout_Zero` geführt (Fendt-Schaltung): Bei einem echten Kommunikationsausfall (`IA_GBSD.TIMEOUT`) wird `0.0` ausgegeben statt des sonst einfach eingefrorenen letzten Werts. Alle Verbindungen sind Adapter (Adapter-Version, analog zu `TECU_GBSD.SUB` im Projekt Krauternter).

Verwendete Funktionsbausteine (FBs)

- **IA_GBSD**: TECU-Eingang GBSD
  - **Typ**: `isobus::tecu::IA_GBSD`
  - **Parameter**: `QI = TRUE`
  - **Erklärung**: Liefert `SPEED` (Rohwert) und `TIMEOUT` (Kommunikationsausfall) als Adapter.
- **Conv_GBSD**: Skalierung des Rohwerts
  - **Typ**: `logiBUS::signalprocessing::fieldbus::AUI_FIELDBUS_UINT_TO_SIGNAL_SCALED`
  - **Parameter**: `SCALE = 0.0036`, `OFFSET = 0`
  - **Erklärung**: Wandelt den `UINT`-Rohwert (`IN`) mit Skalierung und Offset in einen `REAL`-Adapterwert (`OUT`).
- **TZ_GBSD**: Fendt-Schaltung (SubApp)
  - **Typ**: `MyLib::sys::AR_Timeout_Zero` (SubApp-Instanz)
  - **Parameter**: keine
  - **Erklärung**: `OUT` liefert `IN0` unverändert, außer `G` ist `TRUE` (Timeout), dann `0.0`.
- **Q_NumericValue**: numerische Anzeige
  - **Typ**: `isobus::UT::Q::Q_NumericValue_PHYSA`
  - **Parameter**: `stObj = NumberVariable_TECU_Speed_GBSD_N`
  - **Erklärung**: Zeigt den physikalischen Wert aus `rPhys` am UT-Objekt `NumberVariable_TECU_Speed_GBSD_N` an.

Programmablauf und Verbindungen

1. `IA_GBSD.SPEED` ⟶ `Conv_GBSD.IN` (AdapterConnection): Der Rohwert wird skaliert.
2. `Conv_GBSD.OUT` ⟶ `TZ_GBSD.IN0` (AdapterConnection): Der skalierte Wert ist der Live-Wert der Fendt-Schaltung.
3. `IA_GBSD.TIMEOUT` ⟶ `TZ_GBSD.G` (AdapterConnection): Ein Timeout schaltet die Ausgabe auf `0.0`.
4. `TZ_GBSD.OUT` ⟶ `Q_NumericValue.rPhys` (AdapterConnection): Das Ergebnis wird am UT angezeigt.

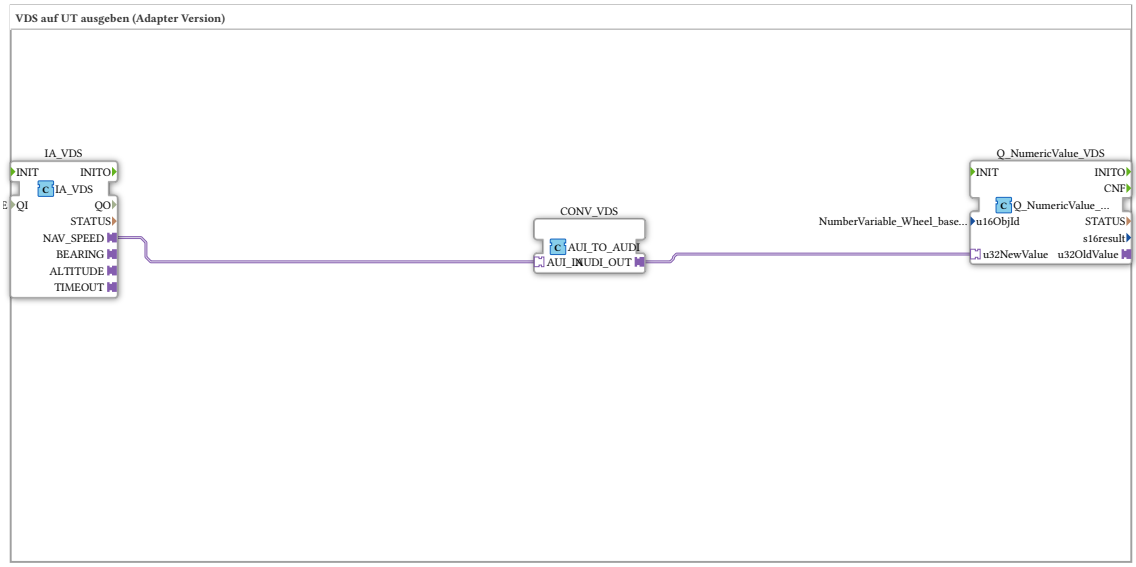
Zusammenfassung

Die Übung 072d_AUI zeigt die Skalierung eines TECU-Rohwerts mit `AUI_FIELDBUS_UINT_TO_SIGNAL_SCALED` und die Absicherung gegen Kommunikationsausfall mit der SubApp `AR_Timeout_Zero`: Fällt die Kommunikation aus, zeigt das UT `0.0` statt eines eingefrorenen Werts.

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

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

2.1.2.353 Uebung_073_AUI: VDS auf UT ausgeben (Adapter Version)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung zeigt, wie ein Geschwindigkeitswert aus dem VDS (Vehicle Data Server) über eine Adapterkonvertierung auf dem UT (Universal Terminal) als numerischer Wert ausgegeben wird. Dabei kommt ein spezieller Adapterbaustein zur Umwandlung des Datentyps von AUI (Application User Interface) nach AUDI (Application User Data Interface) zum Einsatz. Die Konfiguration ist als Subapplikation realisiert.

Verwendete Funktionsbausteine (FBs)

Bausteinname	Typ	Parameter	Kurzbeschreibung
IA_VDS	`isobus::tecu::IA_VDS`	QI = TRUE	Stellt die Verbindung zum VDS her und liefert den Wert für die radbasierte Maschinengeschwindigkeit über den Adapterausgang `NAV_SPEED`.
CONV_VDS	`isobus::conversion::unidirectional::AUI_TO_AUDI`		Konvertiert das AUI-Adapter-Interface in ein AUDI-Inter-

			face, sodass der Datenwert an nachfolgende UT-Bausteine weitergegeben werden kann.
Q_NumericValue_VDS	`isobus::UT::Q::NumericValue_VDI`	`Q_NumericValue_VDI = NumberVariable_Wheel_based_machine_speed`	Zeigt den erhaltenen numerischen Wert auf dem UT-Display an. Die Objekt-ID verweist auf die Variable für die radbasierte Maschinengeschwindigkeit.

Programmablauf und Verbindungen

Die Subapplikation arbeitet in drei Schritten:

1. Der Baustein **IA_VDS** liest kontinuierlich die aktuelle Maschinengeschwindigkeit aus dem VDS. Der Wert wird über den Adapterausgang `NAV_SPEED` (Typ AUI) bereitgestellt.
2. Der **Konverterbaustein** `CONV_VDS` (AUI_TO_AUDI) wandelt das AUI-Interface in ein AUDI-Interface um. Dies ist notwendig, weil der anschließende UT-Baustein einen AUDI-Eingang erwartet.
3. Der konvertierte Wert gelangt über den AUDI-Ausgang `CONV_VDS.AUDI_OUT` zum Dateneingang `u32NewValue` des Bausteins **Q_NumericValue_VDS**. Dieser ist mit der Objekt-ID `NumberVariable_Wheel_based_machine_speed` konfiguriert und zeigt den Geschwindigkeitswert auf dem Universal Terminal an.

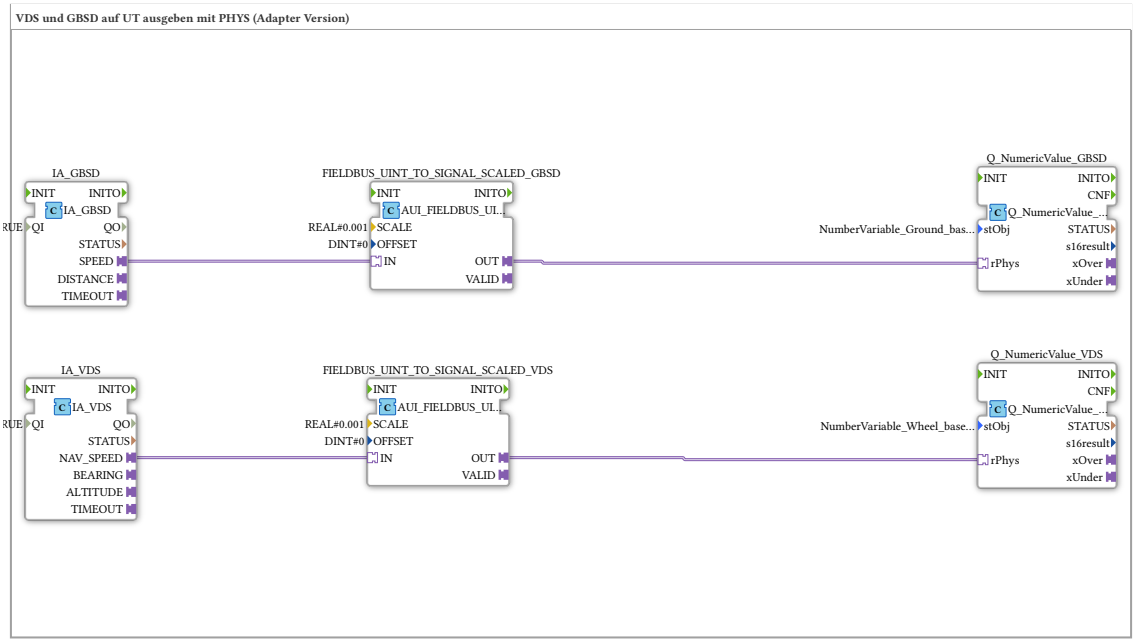
Die folgenden Adapterverbindungen realisieren den Datenfluss:

- `IA_VDS.NAV_SPEED` ⟶ `CONV_VDS.AUI_IN`
- `CONV_VDS.AUDI_OUT` ⟶ `Q_NumericValue_VDS.u32NewValue`

Zusammenfassung

Die Übung veranschaulicht den Einsatz von Adapterbausteinen zur Interface-Konvertierung (AUI ↔ AUDI) innerhalb einer ISOBUS-Anwendung. Durch die klare Trennung von Datenquelle (VDS), Konvertierung und Ausgabe (UT) wird ein modulares und wiederverwendbares Design erreicht. Die Subapplikation kann einfach in übergeordnete Applikationen integriert werden.

2.1.2.354 Uebung_073c_AUI: VDS und GBSD auf UT ausgeben mit PHYS (Adapter Version)



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung demonstriert die Ausgabe der Geschwindigkeitssignale **Ground Based Speed (GBSD)** und **Vehicle/Drive Speed (VDS)** auf einem Universal Terminal (UT) unter Verwendung physikalischer Adressen (PHYS). Die Signale werden über ISOBUS-Adapter (IA) empfangen, skaliert und mittels der `Q_NumericValue_PHYSA`-Bausteine auf dem UT dargestellt. Die Übung vermittelt den Umgang mit Signal-Skalierung und dem Adapter-Konzept (*AUI*) in 4diac IDE.

> **Hinweis:** Derzeit wird als Zielobjekt für die Navigationsgeschwindigkeit ersatzweise `NumberVariable_Wheel_based_machine_speed` verwendet. Für die endgültige Umsetzung sollte `NumberVariable_Navigation_based_vehicle_speed` im Object Pool angelegt und der entsprechende Parameter gesetzt werden.

Verwendete Funktionsbausteine (FBs)

Die Übung besteht aus sechs Funktionsbausteinen, die alle innerhalb der SubApp `Uebung_073c_AUI` angeordnet sind.

Bausteinname	Typ	Parameter	Beschreibung
`IA_GBSD`	`isobus::tecu::IA_GBSD`	QI = TRUE	ISOBUS-Schnittstellenbaustein für die **bodenbezogene Geschwindigkeit** (Ground Based Speed). Liefert den Messwert als UINT am Ausgang `SPEED`.
`IA_VDS`	`isobus::tecu::IA_VDS`	QI = TRUE	ISOBUS-Schnittstellenbaustein für die **fahrzeugbezogene Geschwindigkeit** (Vehicle/Drive Speed). Liefert den Messwert als UINT am Ausgang `NAV_SPEED`.
`FIELDBUS_UINT_TO_SIGNAL_SCALED_GBSD`	`isobus::AUI_FIELDBUS_UINT_TO_SIGNAL_SCALED`	SCALING_SLOT = 0, SET = 0	Skaliert den UINT-Wert von `IA_GBSD` auf einen REAL-Wert (Multiplikation mit 0.001).
`FIELDBUS_UINT_TO_SIGNAL_SCALED_VDS`	`isobus::AUI_FIELDBUS_UINT_TO_SIGNAL_SCALED`	SCALING_SLOT = 0, SET = 0	Skaliert den UINT-Wert von `IA_VDS` auf einen REAL-Wert (Multiplikation mit 0.001).
`Q_NumericValue_GBSD`	`isobus::UT::Q::Q_NumericValue_PHYSA`	sObj = `NumberVariable_Ground_based_machine_speed`	Zeigt die skalierte bodenbezogene Geschwindigkeit auf dem UT an. Verwendet die physikalische Adresse (PHYSA) des Object Pools.

Q_NumericValue_VDS	IA_VDS	:UT::Q::Q_NumericValue_PHYS	NumberVariable_Wheel_based_machine_speed	Zeigt die skalierte fahrzeugbezogene Geschwindigkeit auf dem UT an. *(Hinweis: ersatzweise verwendet)*

Programmablauf und Verbindungen

Die Verbindungen zwischen den Bausteinen erfolgen über **Adapter (AUI)**. Der Datenfluss ist wie folgt:

1. **GBSD**
   - `IA_GBSD` empfängt über den ISOBUS die rohe Geschwindigkeit (UINT).
   - Der Ausgang `SPEED` wird über eine Adapterverbindung an den Eingang `IN` von `FIELDBUS_UINT_TO_SIGNAL_SCALED_GBSD` geleitet.
   - Dieser Baustein skaliert den Wert mit `SCALE = 0.001` und gibt das Ergebnis als REAL am Ausgang `OUT` aus.
   - Der skalierte Wert wird an den Eingang `rPhys` von `Q_NumericValue_GBSD` übergeben und auf dem UT angezeigt.
2. **VDS**
   - `IA_VDS` liefert die Navigationsgeschwindigkeit als UINT am Ausgang `NAV_SPEED`.
   - Dieser Wert wird über eine Adapterverbindung an den skalierenden Baustein `FIELDBUS_UINT_TO_SIGNAL_SCALED_VDS` weitergegeben.
   - Nach der gleichen Skalierung (0.001) wird das Signal an den Eingang `rPhys` von `Q_NumericValue_VDS` geschickt und auf dem UT dargestellt.

Die Skalierung mit `0.001` wandelt die typischerweise ganzzahligen CAN-Bus-Werte (z. B. 0–65535) in physikalische Einheiten (z. B. m/s oder km/h) um. Der Offset ist hier auf 0 gesetzt.

> Der Kommentar im Netzwerk weist darauf hin, dass für die Navigationsgeschwindigkeit eigentlich `NumberVariable_Navigation_based_vehicle_speed` verwendet werden sollte. Die aktuelle Konfiguration nutzt als Platzhalter `NumberVariable_Wheel_based_machine_speed`.

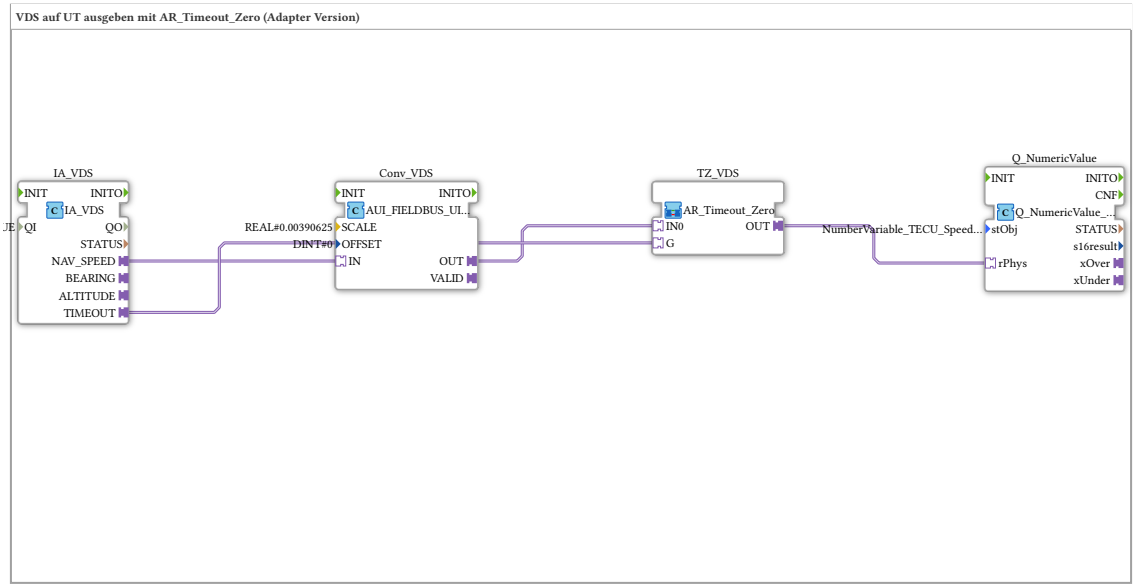
Zusammenfassung

Die Übung **Uebung_073c_AUI** zeigt, wie zwei Geschwindigkeitssignale (GBSD und VDS) über ISOBUS-Adapter (IA) eingelesen, mit einem Faktor von 0.001 skaliert und über die physikalischen Adressen eines Object Pools auf einem Universal Terminal ausgegeben werden. Der Einsatz von Adaptern (AUI) ermöglicht eine flexible Signalverarbeitung ohne feste Punkt-zu-Punkt-Verdrahtung der Ereignisse. Die Übung ist ein typisches Beispiel für die Visualisierung von ISOBUS-Messwerten in landwirtschaftlichen Steuerungen.

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

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

2.1.2.355 Uebung_073d_AUI: VDS auf UT ausgeben mit AR_Timeout_Zero (Adapter Version)



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung gibt die navigationsbasierte Fahrzeuggeschwindigkeit (**VDS**, Navigation-Based Vehicle Speed) des TECU auf einem Universal Terminal (UT) aus. Der Rohwert wird über den ISOBUS-Adapter `IA_VDS` empfangen, mit `0.00390625` (= 1/256) skaliert und über einen physikalischen Wert (`PHYSA`) auf das VT-Zahlenfeld geschrieben. Zusätzlich wird die „Fendt-Schaltung“ `AR_Timeout_Zero` verwendet: Bei einem echten Kommunikationsausfall (Timeout) wird `0.0` angezeigt, statt des sonst einfach eingefrorenen letzten Werts. Die Übung ist analog zu `TECU_VDS.SUB` der Krauternter-Software und verwendet die Adapter-Technik (*AUI*).

Verwendete Funktionsbausteine (FBs)

- **IA_VDS**: ISOBUS-Adapter für Vehicle Direction/Speed
  - **Typ**: `isobus::tecu::IA_VDS`
  - **Parameter**: `QI = TRUE`
  - **Erklärung**: Liefert unter anderem die Navigationsgeschwindigkeit als UINT-Adapter `NAV_SPEED` sowie den Adapter `TIMEOUT` (AX, TRUE bei Kommunikationsausfall). `BEARING` und `ALTITUDE` bleiben in dieser Übung unbenutzt.
- **Conv_VDS**: Skalierung des Rohwerts
  - **Typ**: `logiBUS::signalprocessing::fieldbus::AUI_FIELDBUS_UINT_TO_SIGNAL_SCALED`
  - **Parameter**: `SCALE = 0.00390625`, `OFFSET = 0`
  - **Erklärung**: Wandelt den UINT-Rohwert (`IN`, AUI) in einen skalierten REAL-Wert (`OUT`, AR) um; zusätzlich gibt es einen Adapter `VALID`, der hier nicht verwendet wird.
- **TZ_VDS**: Timeout-Nullung (SubApp-Instanz)
  - **Typ**: `MyLib::sys::AR_Timeout_Zero`
  - **Parameter**: keine
  - **Erklärung**: Liefert `IN0` unverändert an `OUT`, außer `G` ist TRUE (Timeout) - dann wird `0.0` ausgegeben. Intern: `initval_AR` (`INIT_VAL = 0.0`) und der Selektor `AR_AX_SEL_AR`.
- **Q_NumericValue**: Ausgabe auf das VT-Zahlenfeld
  - **Typ**: `isobus::UT::Q::Q_NumericValue_PHYSA`
  - **Parameter**: `stObj = NumberVariable_TECU_Speed_VDS_N`
  - **Erklärung**: Schreibt den physikalischen REAL-Wert (`rPhys`) unter Berücksichtigung von Skalierung, Offset und Nachkommastellen aus `stObj` in das zugehörige Objekt des Object Pools.

Programmablauf und Verbindungen

1. `IA_VDS.NAV_SPEED` ⟶ `Conv_VDS.IN`: der Rohwert wird zur Skalierung geführt.
2. `Conv_VDS.OUT` ⟶ `TZ_VDS.IN0`: der skalierte Wert ist der Live-Wert der Timeout-Nullung.

3. `IA_VDS.TIMEOUT` ⟶ `TZ_VDS.G`: der Kommunikationsstatus entscheidet zwischen Live-Wert und `0.0`.
4. `TZ_VDS.OUT` ⟶ `Q_NumericValue.rPhys`: das Ergebnis wird auf dem UT angezeigt.

Verhalten: Solange Daten vom TECU eintreffen, zeigt das UT die skalierte Fahrzeuggeschwindigkeit. Fällt die Kommunikation aus (`TIMEOUT` = TRUE), springt die Anzeige auf `0.0`.

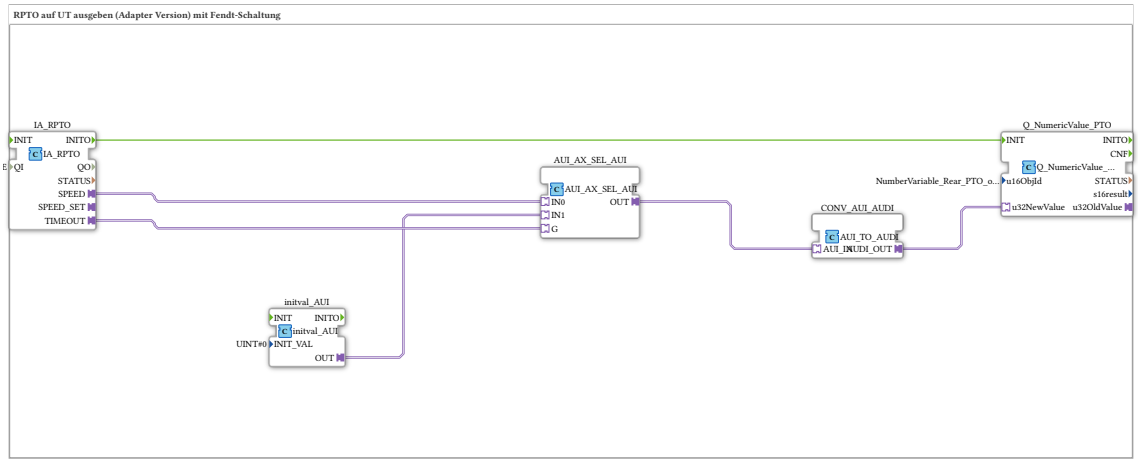
Zusammenfassung

Die Übung zeigt die komplette Signalkette eines TECU-Messwerts im Adapter-Stil: Empfang (`IA_VDS`), Skalierung (`AUI_FIELDBUS_UINT_TO_SIGNAL_SCALED`), Absicherung gegen Kommunikationsausfall (`AR_Timeout_Zero`) und Ausgabe auf dem Terminal (`Q_NumericValue_PHYSA`).

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

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

2.1.2.356 Uebung_074_AUI: RPTO auf UT ausgeben (Adapter Version) mit Fendt-Schaltung



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert die Ausgabe der Drehzahl der hinteren Zapfwelle (Rear PTO) auf ein User Terminal (UT) unter Verwendung von Adaptern. Sie implementiert eine sogenannte „Fendt-Schaltung“: Bei einem Timeout des PTO-Signals wird auf dem UT der Wert 0 angezeigt (Kommentar im Netzwerk). Dadurch wird ein sicheres Verhalten bei Sensorausfall oder Kommunikationsstörungen erreicht.

Die Übung basiert auf ISOBUS-Komponenten (TECU-Interface) und zeigt den Umgang mit Adaptern zur Typumwandlung und Signalauswahl.

Verwendete Funktionsbausteine (FBs)

- **IA_RPTO**: ISOBUS-Adapter für die hintere Zapfwelle
  - **Typ**: `isobus::tecu::IA_RPTO`
  - **Parameter**: `QI = TRUE`
  - **Erklärung**: Schnittstelle zur hinteren Zapfwelle (ISO 11783-7, PGN 65091). Der Adapter `SPEED` liefert die Drehzahl als AUI-Wert, der Adapter `TIMEOUT` meldet den Timeout-Status. Nach der Initialisierung wird `INITO` ausgegeben.
- **initval_AUI**: Ersatzwert 0
  - **Typ**: `adapter::types::unidirectional::AUI::initval::initval_AUI`
  - **Parameter**: `INIT_VAL = 0`
  - **Erklärung**: Stellt den konstanten Wert 0 als AUI-Signal bereit. Er wird bei Timeout statt der Drehzahl angezeigt.
- **AUI_AX_SEL_AUI**: Auswahlbaustein
  - **Typ**: `adapter::iec61131::selection::AUI_AX_SEL_AUI`
  - **Parameter**: keine

  - **Erklärung**: Wählt anhand des AX-Signals `G` zwischen den AUI-Werten `IN0` (bei `G` = FALSE) und `IN1` (bei `G` = TRUE) und gibt den gewählten Wert an `OUT` aus. Dient hier der Realisierung der Fendt-Schaltung.
- **CONV_AUI_AUDI**: Konvertierungsadapter
  - **Typ**: `adapter::conversion::unidirectional::AUI_TO_AUDI`
  - **Parameter**: keine
  - **Erklärung**: Wandelt den AUI-Wert (UINT) in das AUDI-Format (UDINT) um, das der Anzeigebaustein erwartet.
- **Q_NumericValue_PTO**: VT-Anzeige
  - **Typ**: `isobus::UT::Q::Q_NumericValue_AUDI`
  - **Parameter**: `u16ObjId = NumberVariable_Rear_PTO_output_shaft_speed`
  - **Erklärung**: Empfängt den Wert über den Socket `u32NewValue` und zeigt ihn auf dem User Terminal unter der vorgegebenen Objekt-ID an. Wird über das Ereignis `INIT` initialisiert.

Programmablauf und Verbindungen

1. `IA_RPTO.INITO` ⟶ `Q_NumericValue_PTO.INIT` (EventConnection): Nach der Initialisierung des Zapfwellen-Adapters wird die Anzeige initialisiert.
2. `IA_RPTO.SPEED` ⟶ `AUI_AX_SEL_AUI.IN0` (AdapterConnection): Die gemessene Drehzahl liegt am Eingang `IN0` des Selektors an.
3. `initval_AUI.OUT` ⟶ `AUI_AX_SEL_AUI.IN1`: Der Wert 0 liegt am Eingang `IN1` an.
4. `IA_RPTO.TIMEOUT` ⟶ `AUI_AX_SEL_AUI.G`: Der Timeout-Status steuert die Auswahl. Im Normalbetrieb (`TIMEOUT` = FALSE) wird die Drehzahl (`IN0`) durchgeschaltet, bei Timeout (`TIMEOUT` = TRUE) der Wert 0 (`IN1`).
5. `AUI_AX_SEL_AUI.OUT` ⟶ `CONV_AUI_AUDI.AUI_IN`: Der gewählte Wert wird in das AUDI-Format konvertiert.
6. `CONV_AUI_AUDI.AUDI_OUT` ⟶ `Q_NumericValue_PTO.u32NewValue`: Der Wert wird an die Anzeige übergeben und auf dem User Terminal dargestellt - im Normalbetrieb die tatsächliche Drehzahl, bei Timeout eine 0.

Lernziele:

- Verständnis der Verwendung von Adaptern zur Anbindung unterschiedlicher Datentypen (AUI, AUDI).
- Implementierung einer einfachen Fallback-Logik (Fendt-Schaltung) mit einem 2-Kanal-Selektor.
- Umgang mit ISOBUS-TECU-Bausteinen und UT-Anzeigebausteinen in 4diac.
- Erkennen und Behandeln von Timeout-Situationen in der Feldbuskommunikation.

Hinweis zur Einrichtung: Die Objekt-ID `NumberVariable_Rear_PTO_output_shaft_speed` muss im verwendeten ISOBUS-Objektpool definiert sein.

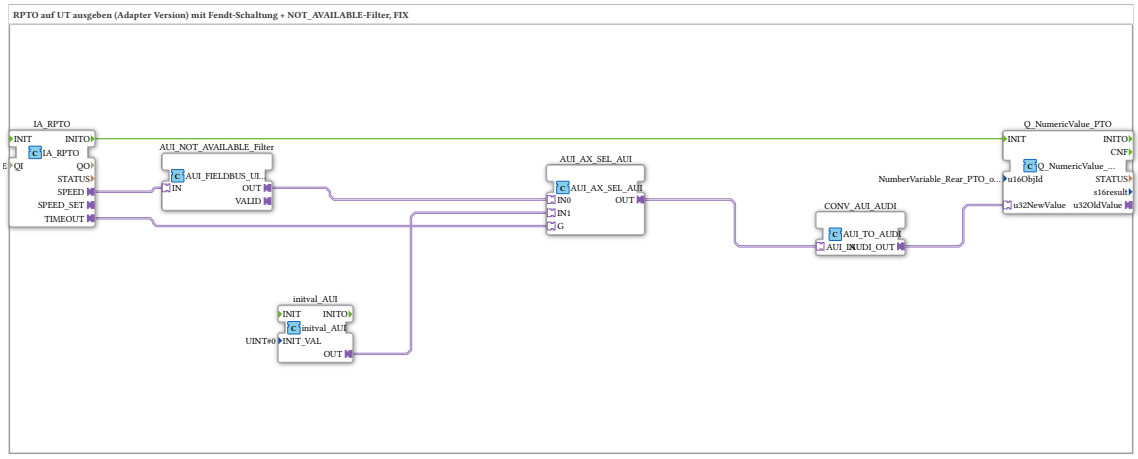
Zusammenfassung

Die Übung `Uebung_074_AUI` realisiert eine sichere Anzeige der Zapfwellendrehzahl auf einem User Terminal. Durch die Kombination eines Adapter-Selektors mit einem konstanten Nullwert im Fehlerfall wird eine einfache, aber robuste Fendt-Schaltung umgesetzt. Sie vermittelt grundlegende Konzepte der Adapter-Kommunikation und der ereignisbasierten Datenvorverarbeitung in 4diac.

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

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

2.1.2.357 Uebung_074a_AUI: RPTO auf UT ausgeben (Adapter-Version) mit Fendt-Schaltung und NOT_AVAILABLE-Filter, FIX



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung gibt die Drehzahl der hinteren Zapfwelle (RPTO) des TECU auf dem ISOBUS-Terminal aus, mit der Fendt-Schaltung (bei Timeout wird 0 angezeigt) und einem zusätzlichen NOT_AVAILABLE-Filter (Version 1.1): Der Filter `AUI_FIELDBUS_UINT_TO_SIGNAL` reagiert schon auf jede einzelne Botschaft, in der ein Fehlwert `0xFFFF` steht, während die Fendt-Schaltung nur den kompletten Kommunikationsausfall (`TIMEOUT`) abdeckt.

Verwendete Funktionsbausteine (FBs)

- **IA_RPTO**: ISOBUS-Adapter für Rear Power Take off (RPTO)
  - **Typ**: `isobus::tecu::IA_RPTO`
  - **Parameter**: `QI = TRUE`
  - **Erklärung**: Liefert die Zapfwellendrehzahl als `AUI`-Adapter `SPEED` und den Timeout-Status als `AX`-Adapter `TIMEOUT`.
- **AUI_NOT_AVAILABLE_Filter**: NOT_AVAILABLE-Filter
  - **Typ**: `logiBUS::signalprocessing::fieldbus::AUI_FIELDBUS_UINT_TO_SIGNAL`
  - **Parameter**: keine
  - **Erklärung**: Spiegelt `IN` nach `OUT`, wenn das Signal gültig ist (`IN` ≤ `VALID_SIGNAL_W`), sonst `OUT` = 0. Damit wird ein Fehlwert `0xFFFF` innerhalb einer Botschaft schon hier auf 0 gemappt.
- **initval_AUI**: Konstante für den Ersatzwert
  - **Typ**: `adapter::types::unidirectional::AUI::initval::initval_AUI`
  - **Parameter**: `INIT_VAL = 0`
  - **Erklärung**: Liefert den festen Ersatzwert `0` als `AUI`-Adapterwert.
- **AUI_AX_SEL_AUI**: Auswahl (Fendt-Schaltung)
  - **Typ**: `adapter::iec61131::selection::AUI_AX_SEL_AUI`
  - **Parameter**: keine
  - **Erklärung**: Wählt über den Adapter `G` zwischen `IN0` (gefilterter Live-Wert) und `IN1` (Ersatzwert `0`). `G` kommt von `IA_RPTO.TIMEOUT`: Bei einem Timeout wird `0` ausgegeben.
- **CONV_AUI_AUDI**: Adapter-Konvertierung
  - **Typ**: `adapter::conversion::unidirectional::AUI_TO_AUDI`
  - **Parameter**: keine
  - **Erklärung**: Wandelt den ausgewählten `AUI`-Wert in das für `Q_NumericValue_AUDI` benötigte `AUDI`-Format um.
- **Q_NumericValue_PTO**: Anzeige auf dem ISOBUS-Terminal
  - **Typ**: `isobus::UT::Q::Q_NumericValue_AUDI`
  - **Parameter**: `u16ObjId = NumberVariable_Rear_PTO_output_shaft_speed`
  - **Erklärung**: Zeigt den Wert im Objekt `NumberVariable_Rear_PTO_output_shaft_speed` an. Das Event `IA_RPTO.INITO` ist mit `Q_NumericValue_PTO.INIT` verbunden.

Programmablauf und Verbindungen

1. `IA_RPTO.INITO` ⟶ `Q_NumericValue_PTO.INIT` (EventConnection): Nach der Initialisierung des TECU-Adapters wird die Anzeige initialisiert.
2. `IA_RPTO.SPEED` ⟶ `AUI_NOT_AVAILABLE_Filter.IN` (AdapterConnection): Die Drehzahl wird zuerst auf Gültigkeit gefiltert.
3. `AUI_NOT_AVAILABLE_Filter.OUT` ⟶ `AUI_AX_SEL_AUI.IN0` und `initval_AUI.OUT` ⟶ `AUI_AX_SEL_AUI.IN1` (AdapterConnections): Gefilterter Live-Wert und Ersatzwert `0` stehen zur Auswahl.
4. `IA_RPTO.TIMEOUT` ⟶ `AUI_AX_SEL_AUI.G` (AdapterConnection): Der Timeout-Status wählt den Ersatzwert, wenn die Kommunikation ausgefallen ist.
5. `AUI_AX_SEL_AUI.OUT` ⟶ `CONV_AUI_AUDI.AUI_IN` und `CONV_AUI_AUDI.AUDI_OUT` ⟶ `Q_NumericValue_PTO.u32NewValue` (AdapterConnections): Der ausgewählte Wert wird gewandelt und angezeigt.

Verhalten: Bei gültigen Botschaften zeigt das Terminal die Zapfwellendrehzahl. Enthält eine Botschaft den Fehlwert `0xFFFF`, wird `0` angezeigt; fällt die Kommunikation ganz aus (`TIMEOUT`), wird ebenfalls `0` angezeigt, statt den letzten Wert einzufrieren.

Zusammenfassung

Die Übung kombiniert zwei Schutzmechanismen für die Anzeige von TECU-Werten: den NOT_AVAILABLE-Filter für Fehlwerte innerhalb einer Botschaft und die Fendt-Schaltung für den Kommunikationsausfall.

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

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

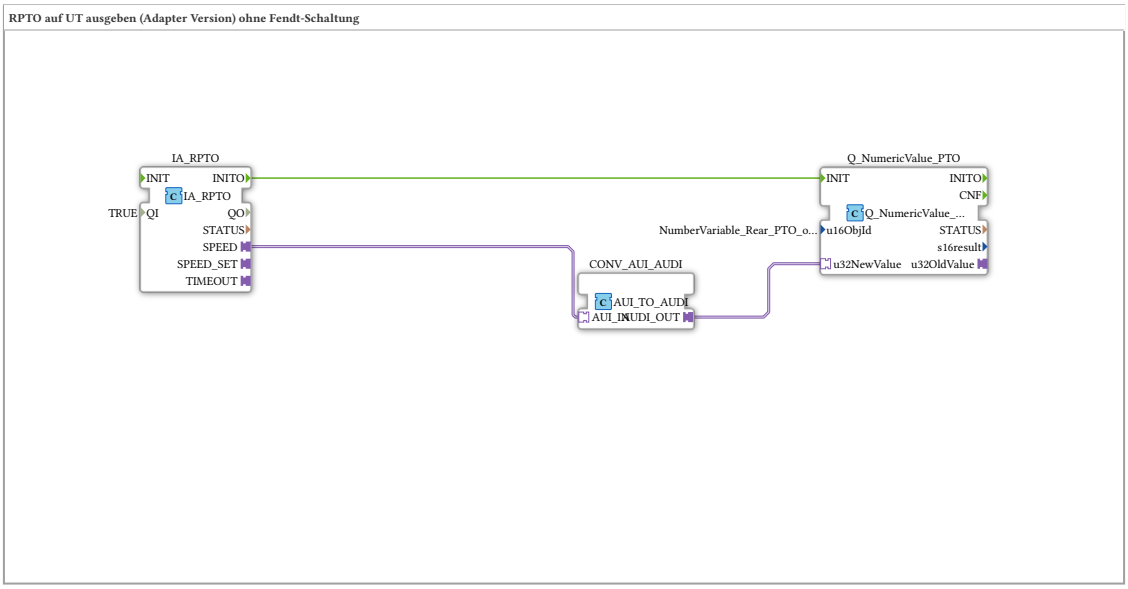
2.1.2.358 Uebung_074b_AUI: RPTO auf UT ausgeben (Adapter Version) ohne Fendt-Schaltung

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_074b_AUI`. Vereinfachte Variante von `Uebung_074a_AUI` (siehe dort für die Fendt-spezifische Ersatzwert-Schaltung): hier wird die Zapfwellendrehzahl ohne Umschaltlogik direkt angezeigt.

Ziel der Übung

Anzeige der Zapfwellendrehzahl (`Rear PTO output shaft speed`) auf dem VT-Terminal, direkt vom TECU-Wert abgeleitet, ohne Fendt-spezifische Ersatzwert-Umschaltung.

Beschreibung und Komponenten

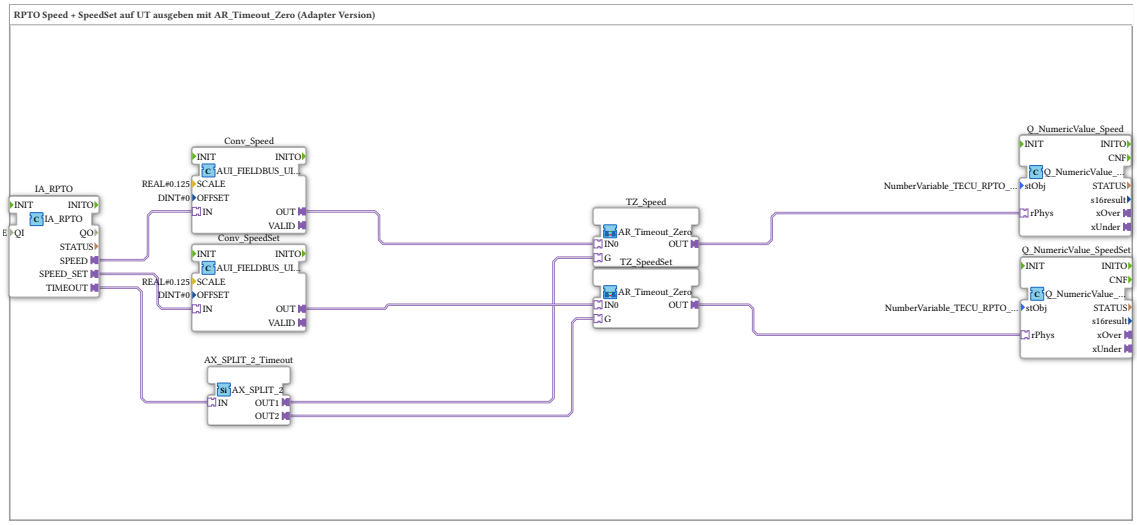


- **IA_RPTO**: `isobus::tecu::IA_RPTO`, liest die Zapfwellendrehzahl vom Tractor-ECU (TECU) über ISOBUS.
- **CONV_AUI_AUDI**: `adapter::conversion::unidirectional::AUI_TO_AUDI`, wandelt den TECU-Wert direkt in das für `Q_NumericValue` benötigte AUDI-Format um.
- **Q_NumericValue_PTO**: zeigt den Wert auf dem VT-Terminal an (`u16ObjId = NumberVariable_Rear_PTO_output_shaft_speed`).

Funktionsweise

1. `IA_RPTO` liest laufend die Zapfwellendrehzahl vom TECU.
2. Der Wert (`SPEED`) wird ohne Auswahl-/Ersatzwertlogik direkt über `CONV_AUI_AUDI` in das Anzeigeformat gewandelt.
3. Bei Initialisierung (`INITO`) wird die Anzeige vorbereitet.

2.1.2.359 Uebung_074c_AUI: RPTO Speed + SpeedSet auf UT ausgeben mit AR_Timeout_Zero (Adapter-Version)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung gibt die Drehzahl (`SPEED`) und den Sollwert (`SPEED_SET`) der hinteren Zapfwelle (RPTO, Rear PTO) des TECU auf dem Universal Terminal (UT) aus. Beide Rohwerte werden mit `SCALE = 0.125` in physikalische `REAL`-Werte skaliert und jeweils durch eine eigene Instanz der SubApp `AR_Timeout_Zero` (Fendt-Schaltung) geführt: Bei einem Kommunikationsausfall (`IA_RPTO.TIMEOUT`) wird `0.0` ausgegeben. Da das Timeout-Signal zwei Ziele hat, wird es über `AX_SPLIT_2` verteilt. Alle Verbindungen sind Adapter (analog zu `TECU_RPTO.SUB` im Projekt Krauternter).

Verwendete Funktionsbausteine (FBs)

- **IA_RPTO**: TECU-Eingang RPTO
  - **Typ**: `isobus::tecu::IA_RPTO`
  - **Parameter**: `QI = TRUE`
  - **Erklärung**: Liefert `SPEED`, `SPEED_SET` und `TIMEOUT` als Adapter.
- **Conv_Speed**, **Conv_SpeedSet**: Skalierung der Rohwerte
  - **Typ**: `logiBUS::signalprocessing::fieldbus::AUI_FIELDBUS_UINT_TO_SIGNAL_SCALED`
  - **Parameter**: jeweils `SCALE = 0.125`, `OFFSET = 0`
  - **Erklärung**: `Conv_Speed` skaliert `SPEED`, `Conv_SpeedSet` skaliert `SPEED_SET`.
- **AX_SPLIT_2_Timeout**: Verteilung des Timeout-Signals
  - **Typ**: `adapter::events::unidirectional::AX_SPLIT_2`
  - **Parameter**: keine
  - **Erklärung**: `OUT1` geht an `TZ_Speed.G`, `OUT2` an `TZ_SpeedSet.G`.
- **TZ_Speed**, **TZ_SpeedSet**: Fendt-Schaltung (SubApps)
  - **Typ**: `MyLib::sys::AR_Timeout_Zero` (SubApp-Instanzen)

  - **Parameter**: keine
  - **Erklärung**: `OUT` liefert `IN0` unverändert, außer `G` ist `TRUE` (Timeout), dann `0.0`.
- **Q_NumericValue_Speed**, **Q_NumericValue_SpeedSet**: numerische Anzeigen
  - **Typ**: `isobus::UT::Q::Q_NumericValue_PHYSA`
  - **Parameter**: `stObj = NumberVariable_TECU_RPTO_Speed_N` bzw. `NumberVariable_TECU_RPTO_SpeedSet_N`
  - **Erklärung**: Zeigen die physikalischen Werte aus `rPhys` an den jeweiligen UT-Objekten an.

Programmablauf und Verbindungen

1. `IA_RPTO.SPEED` → `Conv_Speed.IN` → `TZ_Speed.IN0`, `TZ_Speed.OUT` → `Q_NumericValue_Speed.rPhys` (AdapterConnections): Pfad der Drehzahl.
2. `IA_RPTO.SPEED_SET` → `Conv_SpeedSet.IN` → `TZ_SpeedSet.IN0`, `TZ_SpeedSet.OUT` → `Q_NumericValue_SpeedSet.rPhys` (AdapterConnections): Pfad des Sollwerts.
3. `IA_RPTO.TIMEOUT` → `AX_SPLIT_2_Timeout.IN`, `AX_SPLIT_2_Timeout.OUT1` → `TZ_Speed.G` und `AX_SPLIT_2_Timeout.OUT2` → `TZ_SpeedSet.G` (AdapterConnections): Ein Timeout schaltet beide Ausgaben auf `0.0`.

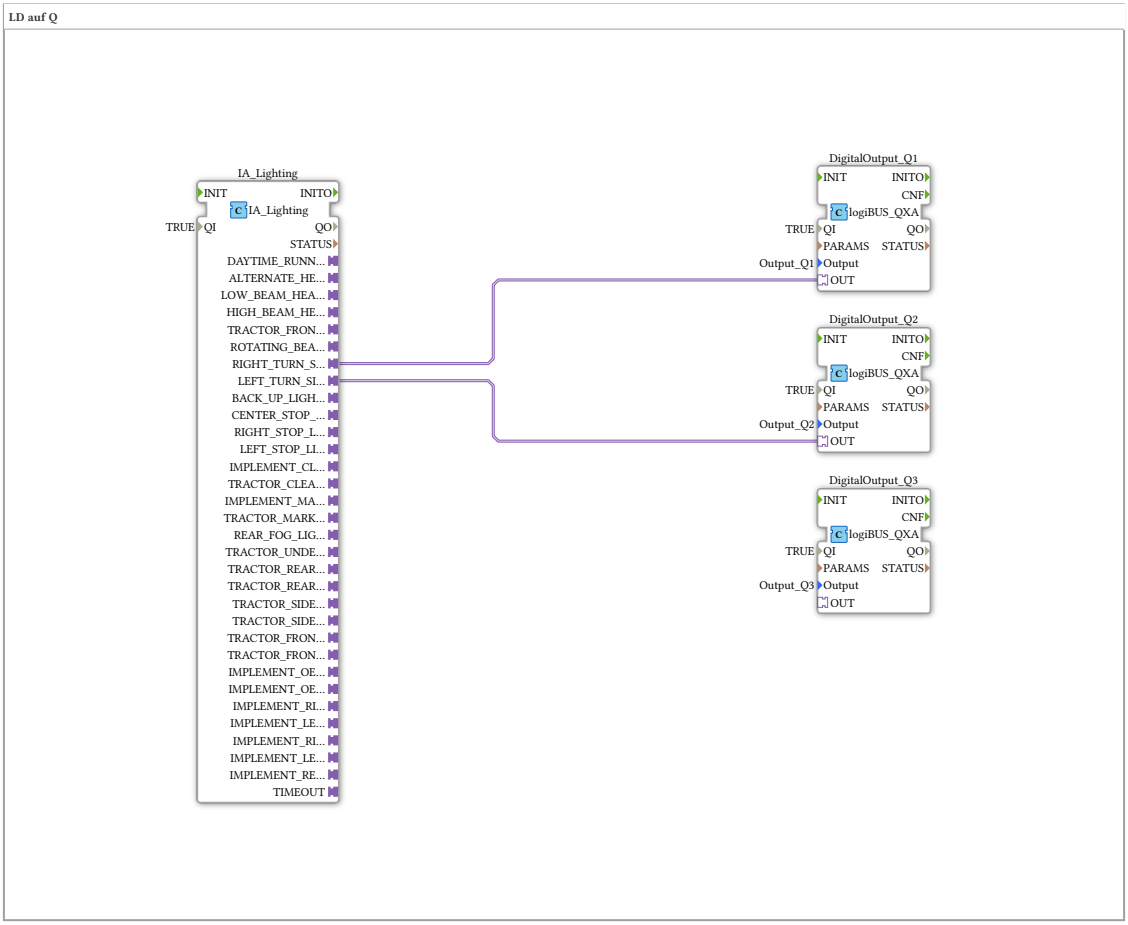
Zusammenfassung

Die Übung 074c_AUI erweitert `Uebung_072d_AUI` auf zwei Werte (Drehzahl und Sollwert der hinteren Zapfwelle). Weil ein Adapter-Plug nur ein Ziel hat, verteilt `AX_SPLIT_2` das gemeinsame Timeout-Signal auf beide `AR_Timeout_Zero`-Instanzen.

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

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

2.1.2.360 Uebung_075_AX: LD auf Q



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert die Ansteuerung digitaler Ausgänge über einen ISOBUS-Lichtadapter. Das SubApp-Element „Uebung_075_AX“ verarbeitet die Blinkersignale (rechts/links) eines Fahrzeugs und leitet sie auf entsprechende digitale Ausgänge (z. B. logiBUS-Ausgänge) weiter. Der Kommentar „LD auf Q“ deutet auf die Übertragung von Lichtdaten (LD) auf die Ausgänge (Q) hin.

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine

IA_Lighting

- **Typ**: `isobus::tecu::IA_Lighting`
- **Verwendete interne FBs**: keine
- **Parameter**:
  - `QI` = TRUE (Freigabe)
- **Funktionsweise**: Dieser Funktionsbaustein stellt ein ISOBUS-konformes Lichtsteuerinterface bereit. Über Adapterausgänge werden verschiedene Lichtfunktionen bereitgestellt, darunter Rechts- und Linksblinker (`RIGHT_TURN_SIGNAL_LIGHTS`, `LEFT_TURN_SIGNAL_LIGHTS`). Die Signale werden aktiv, sobald eine übergeordnete Steuerung die entsprechenden Lichtbefehle sendet.

DigitalOutput_Q1, DigitalOutput_Q2, DigitalOutput_Q3

- **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
- **Verwendete interne FBs**: keine
- **Parameter**:
  - `QI` = TRUE (Freigabe)
  - `Output` = `Output_Q1` / `Output_Q2` / `Output_Q3` (jeweils eigener Wert)
- **Funktionsweise**: Diese Funktionsbausteine kapseln digitale Ausgänge der logiBUS-Hardware. Mit `QI = TRUE` sind sie aktiviert und schalten den angeschlossenen physischen Ausgang gemäß dem ankommenden Adaptersignal. Sie können z. B. für Lampen, Relais oder andere binäre Aktoren verwendet werden.

Programmablauf und Verbindungen

Die Verdrahtung innerhalb des SubApp-Netzwerks erfolgt ausschließlich über **Adapterverbindungen**:

1. Der Baustein `IA_Lighting` empfängt (aus einer übergeordneten Applikation) die Blinkerbefehle und stellt diese an seinen Adapterausgängen bereit.
2. Die Verbindung
   - **`IA_Lighting.RIGHT_TURN_SIGNAL_LIGHTS` ⟶ `DigitalOutput_Q1.OUT`** leitet das Signal für den rechten Blinker zum digitalen Ausgang Q1.
   - **`IA_Lighting.LEFT_TURN_SIGNAL_LIGHTS` ⟶ `DigitalOutput_Q2.OUT`** leitet das Signal für den linken Blinker zum digitalen Ausgang Q2.
3. Der dritte Ausgangsbaustein `DigitalOutput_Q3` bleibt in dieser Übung ungenutzt (kann als Reserve oder für Erweiterungen dienen).

Dank der Adaptertechnik entfällt eine aufwändige Parameterübergabe – die Signalpropagation erfolgt typisiert und automatisch.

Lernziele:

- Verständnis von Adapterverbindungen (Socket/Plug) in 4diac IDE
- Einbindung eines ISOBUS-Lichtadapters und logiBUS-Digitalausgängen
- Erstellen wiederverwendbarer SubApp-Komponenten für Fahrzeuglichtsteuerungen

Schwierigkeitsgrad: Einfach (grundlegende Adapterkonfiguration) **Vorkenntnisse**: Grundlagen der 4diac-IDE, Umgang mit Funktionsbausteinen und Netzwerken

Zusammenfassung

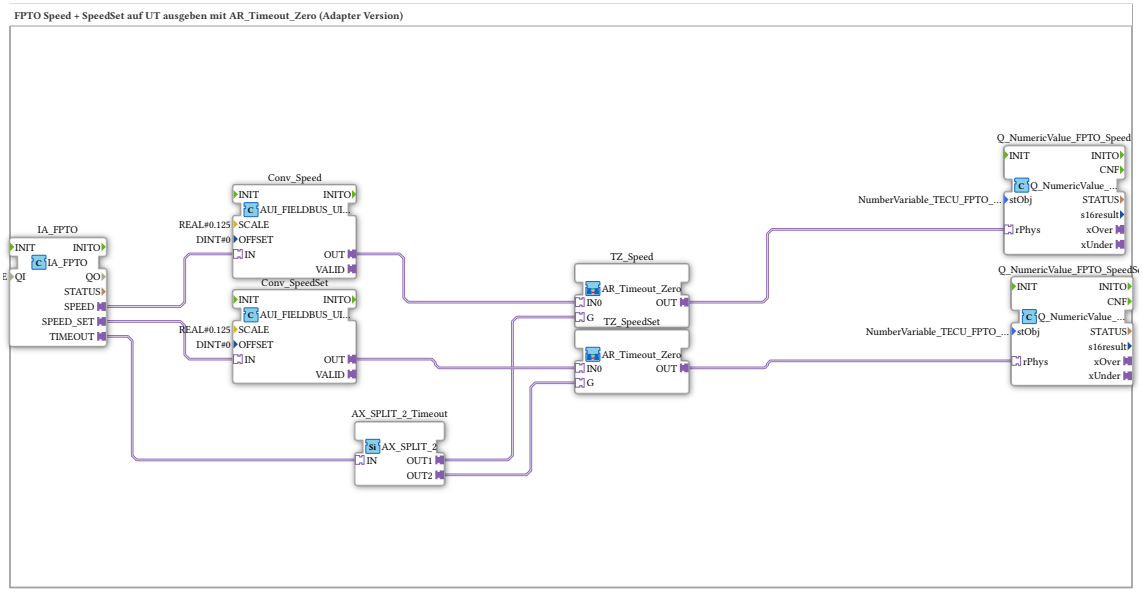
Die Übung „Uebung_075_AX“ zeigt, wie ein ISOBUS-Lichtadapter über Adapterverbindungen zwei digitale Ausgänge ansteuert. Rechts- und Linksblinksignale werden auf die logiBUS-Ausgänge Q1 und Q2 übertragen. Die SubApp ist ein kompaktes, prototypisches Element für einfache Lichtsteuerungen in

landwirtschaftlichen Fahrzeugen. Sie verdeutlicht die Vorteile der adapterbasierten Kommunikation in der IEC 61499 und kann als Grundbaustein für komplexere Lichtfunktionen dienen.

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

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

2.1.2.361 Uebung_075a_AUI: FPTO Speed + SpeedSet auf UT ausgeben mit AR_Timeout_Zero (Adapter Version)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung gibt die Drehzahl (**Speed**) und die Solldrehzahl (**SpeedSet**) der Front-Zapfwelle (FPTO, Front Power Take-off) des TECU auf einem Universal Terminal (UT) aus. Beide Rohwerte kommen über den Adapter `IA_FPTO`, werden mit `0.125` skaliert und über `Q_NumericValue_PHYSA` angezeigt. Für jeden Wert sorgt eine Instanz der „Fendt-Schaltung“ `AR_Timeout_Zero` dafür, dass bei einem Kommunikationsausfall `0.0` statt des eingefrorenen letzten Werts angezeigt wird. Die Übung ist analog zu `TECU_FPTO.SUB` der Krauternter-Software.

Verwendete Funktionsbausteine (FBs)

- **IA_FPTO**: ISOBUS-Adapter für die Front-Zapfwelle
  - **Typ**: `isobus::tecu::IA_FPTO`
  - **Parameter**: `QI = TRUE`
  - **Erklärung**: Liefert die UINT-Adapter `SPEED` (Drehzahl der Abtriebswelle) und `SPEED_SET` (Soll-Drehzahl) sowie den Adapter `TIMEOUT` (AX, TRUE bei Kommunikationsausfall).
- **AX_SPLIT_2_Timeout**: Verteiler für den Timeout-Status
  - **Typ**: `adapter::events::unidirectional::AX_SPLIT_2`
  - **Parameter**: keine
  - **Erklärung**: Verteilt `IA_FPTO.TIMEOUT` auf `OUT1` (⟶ `TZ_Speed.G`) und `OUT2` (⟶ `TZ_SpeedSet.G`).
- **Conv_Speed**, **Conv_SpeedSet**: Skalierung der Rohwerte
  - **Typ**: `logiBUS::signalprocessing::fieldbus::AUI_FIELDBUS_UINT_TO_SIGNAL_SCALED`
  - **Parameter**: je `SCALE = 0.125`, `OFFSET = 0`
  - **Erklärung**: Wandeln den UINT-Rohwert (`IN`) in einen skalierten REAL-Wert (`OUT`, AR) um.
- **TZ_Speed**, **TZ_SpeedSet**: Timeout-Nullung (SubApp-Instanzen)
  - **Typ**: `MyLib::sys::AR_Timeout_Zero`
  - **Parameter**: keine
  - **Erklärung**: Liefern `IN0` unverändert an `OUT`, außer `G` ist TRUE (Timeout) - dann `0.0`.
- **Q_NumericValue_FPTO_Speed**: Ausgabe der Drehzahl

  - **Typ**: `isobus::UT::Q::Q_NumericValue_PHYSA`
  - **Parameter**: `stObj = NumberVariable_TECU_FPTO_Speed_N`
  - **Erklärung**: Schreibt den physikalischen Wert (`rPhys`) in das zugehörige VT-Objekt.
- **Q_NumericValue_FPTO_SpeedSet**: Ausgabe der Solldrehzahl
  - **Typ**: `isobus::UT::Q::Q_NumericValue_PHYSA`
  - **Parameter**: `stObj = NumberVariable_TECU_FPTO_SpeedSet_N`
  - **Erklärung**: Schreibt den physikalischen Wert (`rPhys`) in das zugehörige VT-Objekt.

Programmablauf und Verbindungen

1. Drehzahl: `IA_FPTO.SPEED` ⟶ `Conv_Speed.IN` ⟶ `TZ_Speed.IN0`; `AX_SPLIT_2_Timeout.OUT1` ⟶ `TZ_Speed.G`; `TZ_Speed.OUT` ⟶ `Q_NumericValue_FPTO_Speed.rPhys`.
2. Solldrehzahl: `IA_FPTO.SPEED_SET` ⟶ `Conv_SpeedSet.IN` ⟶ `TZ_SpeedSet.IN0`; `AX_SPLIT_2_Timeout.OUT2` ⟶ `TZ_SpeedSet.G`; `TZ_SpeedSet.OUT` ⟶ `Q_NumericValue_FPTO_SpeedSet.rPhys`.
3. `IA_FPTO.TIMEOUT` ⟶ `AX_SPLIT_2_Timeout.IN`: ein einziger Timeout-Status steuert beide Nullungen.

Verhalten: Solange Daten eintreffen, zeigt das UT Drehzahl und Solldrehzahl (je Rohwert x 0.125). Bei Kommunikationsausfall (`TIMEOUT` = TRUE) zeigen beide Felder `0.0`.

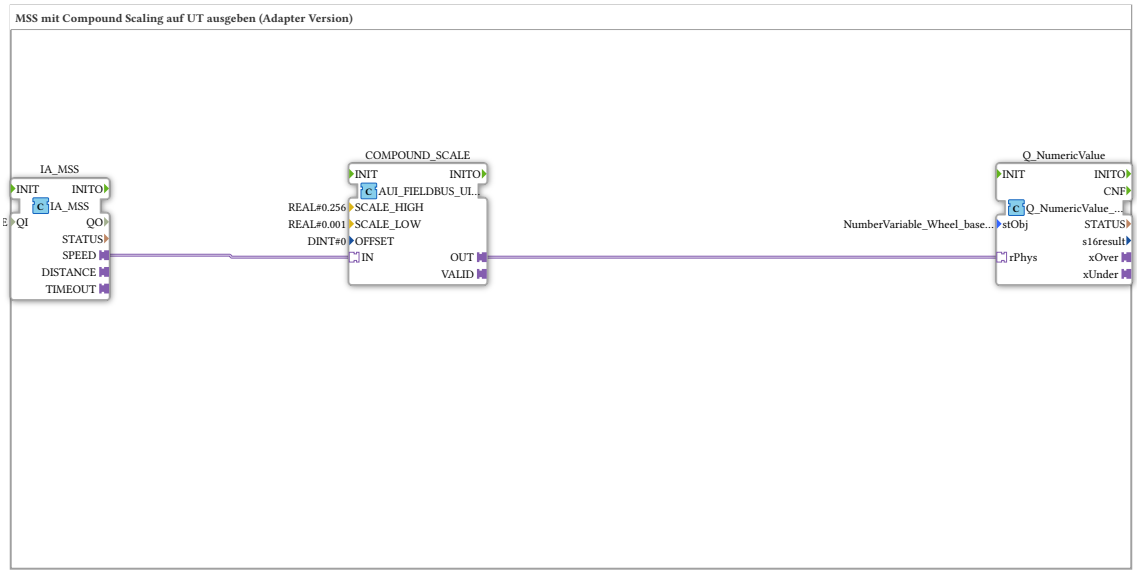
Zusammenfassung

Die Übung erweitert die Signalkette aus `Uebung_073d_AUI` auf zwei Messwerte desselben ISOBUS-Adapters: Der gemeinsame Timeout-Status wird mit `AX_SPLIT_2` auf zwei `AR_Timeout_Zero`-Instanzen verteilt, sodass beide Anzeigen konsistent auf `0.0` fallen.

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

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

2.1.2.362 Uebung_076_AUI: MSS mit Compound Scaling auf UT ausgeben (Adapter Version)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert die Verarbeitung der Machine Selected Speed (MSS) und deren Weiterleitung als physikalische Größe auf das Universal Terminal (UT). Dabei kommt ein Compound Scaling (zusammengesetzte Skalierung) zum Einsatz, um den Wertebereich der Signale an die Anforderungen des UT anzupassen. Die Kommunikation zwischen den Funktionsbausteinen erfolgt über Adapterverbindungen.

Verwendete Funktionsbausteine (FBs)

IA_MSS

- **Typ**: `isobus::tecu::IA_MSS`
- **Parameter**: QI = TRUE
- **Funktionsweise**: Dieser FB stellt die Schnittstelle zur ISO-bus-basierten Maschinensteuerung dar. Er liefert die aktuelle Maschinensollgeschwindigkeit (MSS) als Ausgangssignal am Adapterport `SPEED`. Der Parameter QI (Qualifier Input) wird fest auf TRUE gesetzt, um die Datenbereitstellung zu aktivieren.

COMPOUND_SCALE

- **Typ**: `logiBUS::signalprocessing::fieldbus::AUI_FIELDBUS_UINT_TO_SIGNAL_COMPOUND_SCALE`
- **Parameter**:
  - `SCALE_HIGH` = REAL#0.256
  - `SCALE_LOW` = REAL#0.001
  - `OFFSET` = DINT#0
- **Funktionsweise**: Dieser FB empfängt den Rohwert (UINT) vom IA_MSS über den Adaptereingang `IN` und skaliert ihn mittels einer zusammengesetzten Skalierung (Compound Scaling) in einen physikalischen Wert. Die Skalierungsparameter legen die obere und untere Grenze des linearen Bereichs fest; der Offset bleibt hier Null. Der skalierte Wert wird am Ausgang `OUT` bereitgestellt.

Q_NumericValue

- **Typ**: `isobus::UT::Q::Q_NumericValue_PHYSA`
- **Parameter**:
  - `stObj` = `NumberVariable_Wheel_based_machine_speed`
- **Funktionsweise**: Dieser FB dient der Ausgabe eines physikalischen Werts auf das Universal Terminal (UT) über die ISO-bus-UT-Schnittstelle. Der Parameter `stObj` referenziert ein Objekt aus dem Object Pool (hier: `NumberVariable_Wheel_based_machine_speed`), das als Zielvariable für die Wertanzeige dient. Der skalierte Wert wird über den Adaptereingang `rPhys` übernommen.

> **Hinweis:** Der verwendete Object-Pool-Eintrag ist nur ein Platzhalter. In der Praxis sollte ein eigenes `NumberVariable_Machine_selected_speed`-Objekt angelegt und der Parameter entsprechend geändert werden (siehe Kommentar im Netzwerk).

Programmablauf und Verbindungen

Der Ablauf ist wie folgt:

1. Der Funktionsbaustein `IA_MSS` liefert die aktuelle Maschinensollgeschwindigkeit als UINT-Wert über seinen Adapterausgang `SPEED`.
2. Dieser Wert wird über eine Adapterverbindung an den Eingang `IN` des Funktionsbausteins `COMPOUND_SCALE` weitergeleitet.
3. `COMPOUND_SCALE` führt die zusammengesetzte Skalierung durch und gibt den resultierenden physikalischen Wert (Realer Wert) am Ausgang `OUT` aus.
4. Der skalierte Wert wird über eine weitere Adapterverbindung an den Eingang `rPhys` des Funktionsbausteins `Q_NumericValue` übergeben.
5. `Q_NumericValue` schreibt den Wert in das referenzierte Object-Pool-Objekt, sodass er auf dem UT angezeigt werden kann.

Die gesamte Kommunikation zwischen den Blöcken erfolgt über Adapter (keine direkten Daten- oder Ereignisverbindungen), was den modularen Aufbau und die Wiederverwendbarkeit erleichtert.

Zusammenfassung

Die Übung veranschaulicht die typische ISO-bus-Datenverarbeitungskette: **Sensor/MSS → Skalierung → Ausgabe auf Terminal**. Durch die Verwendung von Adapterverbindungen bleibt die Konfiguration flexibel und erweiterbar. Lernziele sind unter anderem:

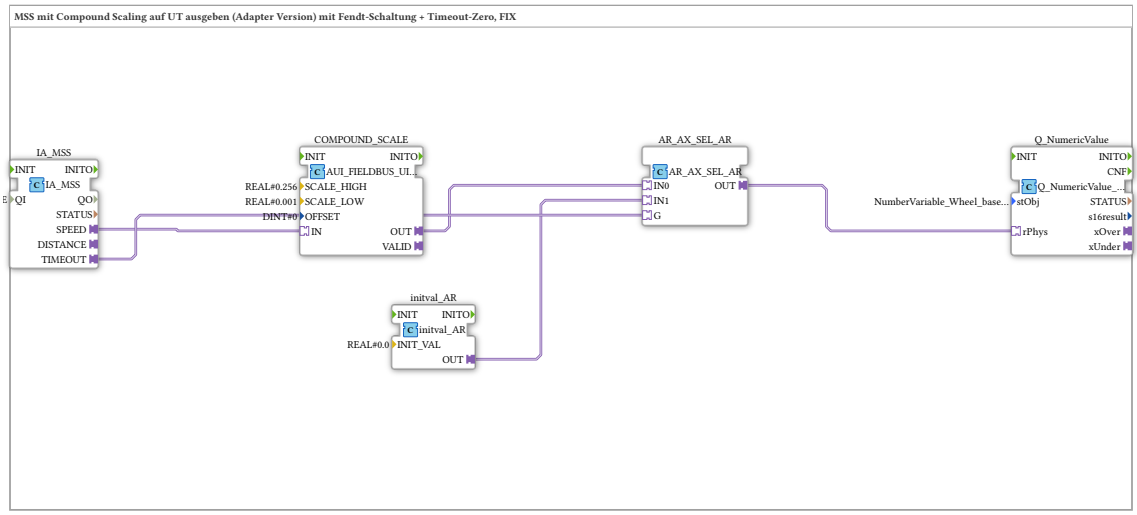
- Verständnis der MSS-Verarbeitung in der Landtechnik
- Anwendung von Compound Scaling (zusammengesetzte Skalierung)
- Arbeiten mit ISO-bus-Adapter-FBs und Object-Pool-Referenzen

Die Übung erfordert Grundkenntnisse in der 4diac-IDE und im Umgang mit ISO-bus-Funktionsbausteinen.

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

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

2.1.2.363 Uebung_076a_AUI: MSS mit Compound Scaling auf UT ausgeben (Adapter Version) mit Fendt-Schaltung + Timeout-Zero



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung gibt die maschinenselektierte Geschwindigkeit (Machine Selected Speed, MSS) der TECU auf einem User Terminal (UT) aus. Der 16-Bit-Rohwert des ISOBUS-Adapters wird mit einer zusammengesetzten Skalierung (Compound Scaling, getrennte Faktoren für oberes und unteres Byte) in einen physikalischen REAL-Wert umgerechnet. Zusätzlich ist die „Fendt-Schaltung“ mit Timeout-Zero umgesetzt: Bei einem Timeout wird auf dem UT der Wert 0.0 angezeigt (Kommentar im Netzwerk: analog zur Krauternter-`TECU_MSS.SUB`). Alle Verbindungen laufen über Adapter.

Verwendete Funktionsbausteine (FBs)

- **IA_MSS**: ISOBUS-Adapter für die maschinenselektierte Geschwindigkeit
  - **Typ**: `isobus::tecu::IA_MSS`
  - **Parameter**: `QI = TRUE`
  - **Erklärung**: Stellt die Geschwindigkeit als AUI-Wert am Adapter `SPEED` und den Timeout-Status am Adapter `TIMEOUT` bereit (PGN 61474).
- **COMPOUND_SCALE**: zusammengesetzte Skalierung
  - **Typ**: `logiBUS::signalprocessing::fieldbus::AUI_FIELDBUS_UINT_TO_SIGNAL_COMPOUND_SCALE`
  - **Parameter**: `SCALE_HIGH = REAL#0.256`, `SCALE_LOW = REAL#0.001`, `OFFSET = DINT#0`
  - **Erklärung**: Teilt den 16-Bit-Eingangswert in oberes und unteres Byte, skaliert beide mit `SCALE_HIGH` bzw. `SCALE_LOW`, summiert sie und addiert den `OFFSET`. Das Ergebnis liegt am AR-Adapter `OUT`.
- **initval_AR**: Ersatzwert 0.0
  - **Typ**: `adapter::types::unidirectional::AR::initval::initval_AR`
  - **Parameter**: `INIT_VAL = REAL#0.0`
  - **Erklärung**: Stellt den konstanten Wert 0.0 als AR-Signal bereit; er wird bei Timeout statt der Geschwindigkeit angezeigt.
- **AR_AX_SEL_AR**: Auswahlbaustein
  - **Typ**: `adapter::iec61131::selection::AR_AX_SEL_AR`
  - **Parameter**: keine
  - **Erklärung**: Wählt anhand des AX-Signals `G` zwischen den AR-Werten `IN0` (`G` = FALSE) und `IN1` (`G` = TRUE) und gibt den gewählten Wert an `OUT` aus.
- **Q_NumericValue**: VT-Anzeige

- **Typ**: `isobus::UT::Q::Q_NumericValue_PHYSA`
- **Parameter**: `stObj = NumberVariable_Wheel_based_machine_speed_N`
- **Erklärung**: Zeigt den physikalischen REAL-Wert vom Adapter `rPhys` auf dem im Objektpool-Eintrag `stObj` angegebenen VT-Objekt an. Laut Kommentar im Netzwerk wird ersatzweise `NumberVariable_Wheel_based_machine_speed` verwendet, solange `NumberVariable_Machine_selected_speed` im Objektpool noch nicht angelegt ist (TODO).

Programmablauf und Verbindungen

1. `IA_MSS.SPEED` ⟶ `COMPOUND_SCALE.IN` (AdapterConnection): Der Rohwert der Geschwindigkeit wird skaliert.
2. `COMPOUND_SCALE.OUT` ⟶ `AR_AX_SEL_AR.IN0`: Der skalierte Wert ist der Live-Wert (`IN0`).
3. `initval_AR.OUT` ⟶ `AR_AX_SEL_AR.IN1`: Der Wert 0.0 liegt am Eingang `IN1` an.
4. `IA_MSS.TIMEOUT` ⟶ `AR_AX_SEL_AR.G`: Der Timeout-Status steuert die Auswahl. Im Normalbetrieb (`TIMEOUT` = FALSE) wird die Geschwindigkeit durchgeschaltet, bei Timeout (`TIMEOUT` = TRUE) der Wert 0.0.
5. `AR_AX_SEL_AR.OUT` ⟶ `Q_NumericValue.rPhys`: Der gewählte Wert wird auf dem UT angezeigt.

Verhalten: Im Normalbetrieb zeigt das UT die skalierte maschinenselektierte Geschwindigkeit, bei Timeout der TECU-Nachricht 0.0.

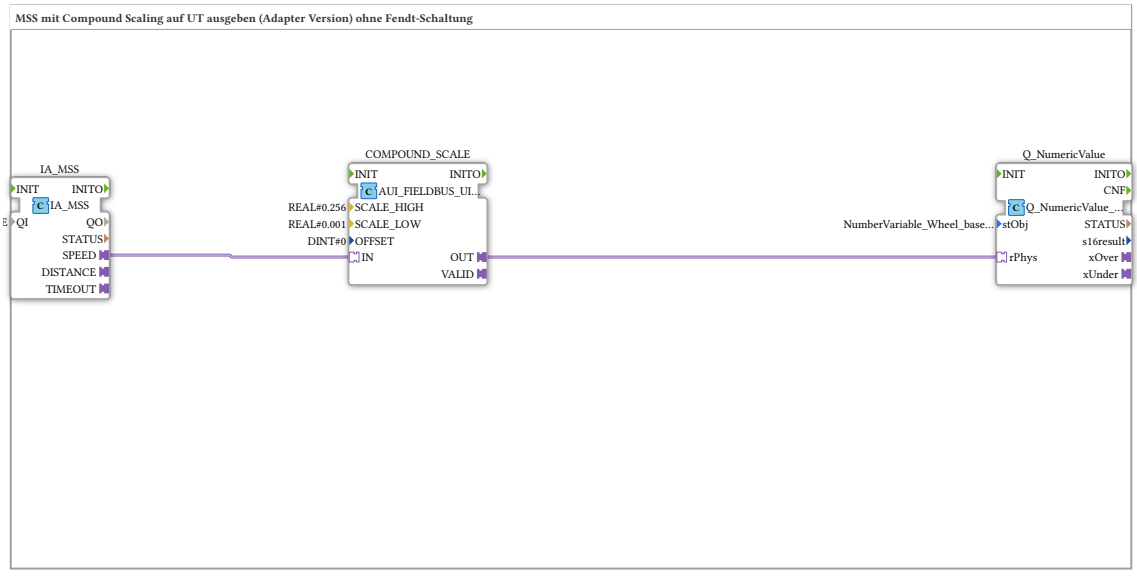
Zusammenfassung

Die Übung kombiniert eine zusammengesetzte Skalierung des 16-Bit-TECU-Rohwerts mit der Fendt-Schaltung: Ein Adapter-Selektor ersetzt bei Timeout den Messwert durch 0.0, sodass die Anzeige bei Kommunikationsausfall nicht auf dem letzten Wert stehen bleibt.

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

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

2.1.2.364 Uebung_076b_AUI: MSS mit Compound Scaling auf UT ausgeben (Adapter-Version) ohne Fendt-Schaltung



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung gibt die maschinenausgewählte Geschwindigkeit (Machine Selected Speed, MSS) des TECU auf dem ISOBUS-Terminal aus. Der `AUI`-Wert wird mit Compound Scaling (getrennte Faktoren für oberes und unteres Byte) in einen physikalischen Wert gewandelt. Es gibt keine Fendt-Schaltung: Bei einem Timeout wird nicht auf `0.0` umgeschaltet. Die Übung ist die Adapter-Version von `Uebung_076` ohne Fendt-Schaltung.

Verwendete Funktionsbausteine (FBs)

- **IA_MSS**: ISOBUS-Adapter für Machine Selected Speed (MSS)
  - **Typ**: `isobus::tecu::IA_MSS`
  - **Parameter**: `QI = TRUE`
  - **Erklärung**: Liefert die maschinenausgewählte Geschwindigkeit als `AUI`-Adapter `SPEED`.
- **COMPOUND_SCALE**: Compound-Skalierung
  - **Typ**: `logiBUS::signalprocessing::fieldbus::AUI_FIELDBUS_UINT_TO_SIGNAL_COMPOUND_SCALE`
  - **Parameter**: `SCALE_HIGH = REAL#0.256`, `SCALE_LOW = REAL#0.001`, `OFFSET = DINT#0`
  - **Erklärung**: Wandelt den `AUI`-Wert nur bei gültigem Signal mit getrennten Skalierungsfaktoren für oberes (`SCALE_HIGH`) und unteres Byte (`SCALE_LOW`) in einen `AR`-Wert um; `OFFSET` wird nach der Skalierung addiert.
- **Q_NumericValue**: Anzeige auf dem ISOBUS-Terminal
  - **Typ**: `isobus::UT::Q::Q_NumericValue_PHYSA`
  - **Parameter**: `stObj = NumberVariable_Wheel_based_machine_speed_N`
  - **Erklärung**: Zeigt den physikalischen Wert (`rPhys`) im Objekt `NumberVariable_Wheel_based_machine_speed_N` an. Laut Netzwerk-Kommentar ist das ein Ersatz: `NumberVariable_Machine_selected_speed` muss noch im Object Pool angelegt und hier eingetragen werden.

Programmablauf und Verbindungen

1. `IA_MSS.SPEED` ⟶ `COMPOUND_SCALE.IN` (AdapterConnection): Der Geschwindigkeitswert vom TECU wird skaliert.
2. `COMPOUND_SCALE.OUT` ⟶ `Q_NumericValue.rPhys` (AdapterConnection): Der skalierte Wert wird auf dem Terminal angezeigt.

Verhalten: Das Terminal zeigt den skalierten MSS-Wert an. Der Timeout-Adapter `IA_MSS.TIMEOUT` ist nicht verdrahtet.

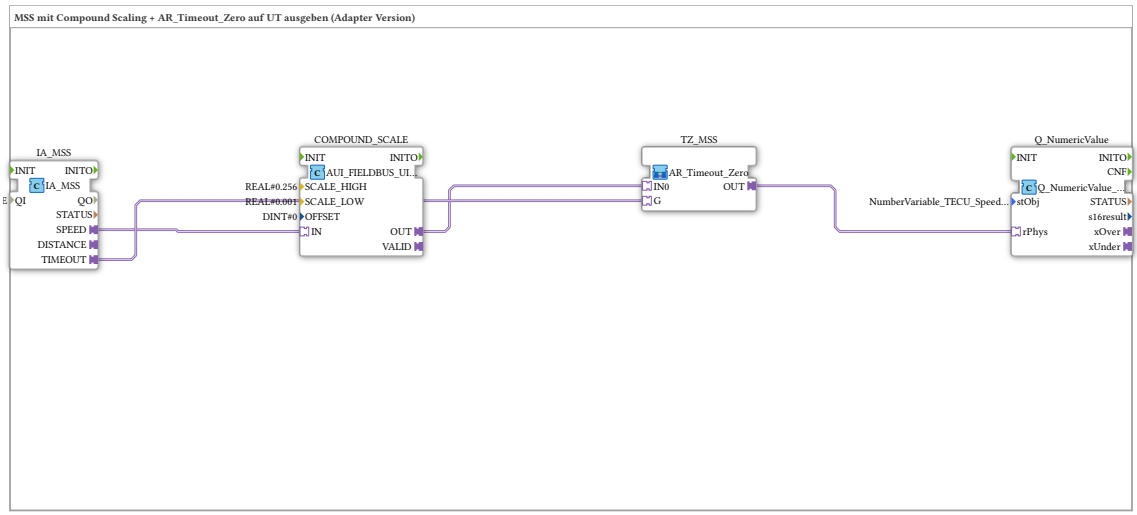
Zusammenfassung

Die Übung zeigt die kürzeste Kette für einen TECU-Wert mit zweistufiger Skalierung: TECU-Adapter ⟶ Compound Scaling ⟶ Anzeige. Der noch fehlende Objekt-Pool-Eintrag `NumberVariable_Machine_selected_speed` ist im Netzwerk als TODO vermerkt.

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

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

2.1.2.365 Uebung_076c_AUI: MSS mit Compound Scaling + AR_Timeout_Zero auf UT ausgeben (Adapter-Version)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung gibt die MSS-Geschwindigkeit des TECU auf dem Universal Terminal (UT) aus. Der Rohwert wird mit einer Compound-Skalierung (`SCALE_HIGH = 0.256`, `SCALE_LOW = 0.001`, `OFFSET = 0`) in einen physikalischen `REAL`-Wert gewandelt und durch die SubApp `AR_Timeout_Zero` (Fendt-Schaltung) geführt: Bei einem Kommunikationsausfall (`IA_MSS.TIMEOUT`) wird `0.0` ausgegeben. Alle Verbindungen sind Adapter (analog zu `TECU_MSS.SUB` im Projekt Krauternter).

Verwendete Funktionsbausteine (FBs)

- **IA_MSS**: TECU-Eingang MSS
  - **Typ**: `isobus::tecu::IA_MSS`
  - **Parameter**: `QI = TRUE`
  - **Erklärung**: Liefert `SPEED` (Rohwert) und `TIMEOUT` (Kommunikationsausfall) als Adapter.
- **COMPOUND_SCALE**: Compound-Skalierung
  - **Typ**: `logiBUS::signalprocessing::fieldbus::AUI_FIELDBUS_UINT_TO_SIGNAL_COMPOUND_SCALE`
  - **Parameter**: `SCALE_HIGH = REAL#0.256`, `SCALE_LOW = REAL#0.001`, `OFFSET = DINT#0`
  - **Erklärung**: Wandelt den `UINT`-Rohwert (`IN`) mit den zwei Skalierungsfaktoren `SCALE_HIGH` und `SCALE_LOW` in einen `REAL`-Adapterwert (`OUT`).
- **TZ_MSS**: Fendt-Schaltung (SubApp)
  - **Typ**: `MyLib::sys::AR_Timeout_Zero` (SubApp-Instanz)
  - **Parameter**: keine
  - **Erklärung**: `OUT` liefert `IN0` unverändert, außer `G` ist `TRUE` (Timeout), dann `0.0`.
- **Q_NumericValue**: numerische Anzeige
  - **Typ**: `isobus::UT::Q::Q_NumericValue_PHYSA`
  - **Parameter**: `stObj = NumberVariable_TECU_Speed_MSS_N`
  - **Erklärung**: Zeigt den physikalischen Wert aus `rPhys` am UT-Objekt `NumberVariable_TECU_Speed_MSS_N` an.

Programmablauf und Verbindungen

1. `IA_MSS.SPEED` → `COMPOUND_SCALE.IN` (AdapterConnection): Der Rohwert wird mit Compound-Skalierung gewandelt.
2. `COMPOUND_SCALE.OUT` → `TZ_MSS.IN0` (AdapterConnection): Der skalierte Wert ist der Live-Wert der Fendt-Schaltung.
3. `IA_MSS.TIMEOUT` → `TZ_MSS.G` (AdapterConnection): Ein Timeout schaltet die Ausgabe auf `0.0`.
4. `TZ_MSS.OUT` → `Q_NumericValue.rPhys` (AdapterConnection): Das Ergebnis wird am UT angezeigt.

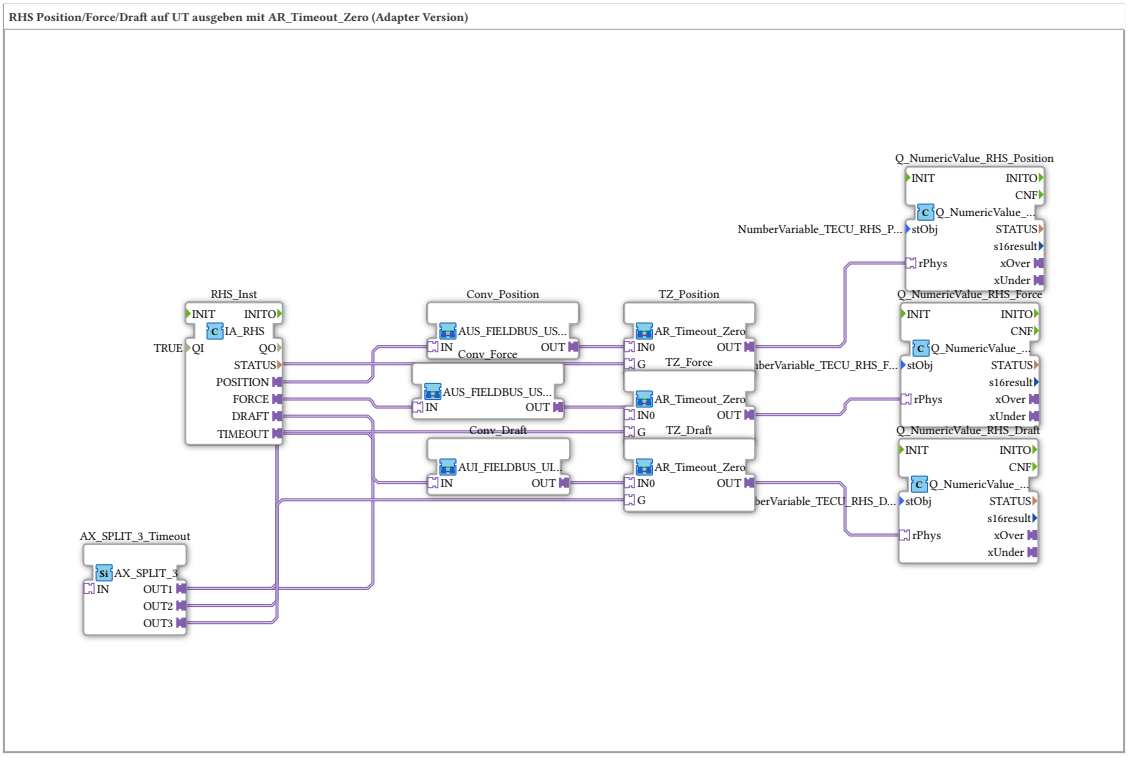
Zusammenfassung

Die Übung 076c_AUI entspricht in der Struktur `Uebung_072d_AUI`, verwendet aber statt der einfachen Skalierung den Baustein `AUI_FIELDBUS_UINT_TO_SIGNAL_COMPOUND_SCALE` mit zwei Skalierungsfaktoren. Die Absicherung gegen Kommunikationsausfall übernimmt wieder `AR_Timeout_Zero`.

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

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

2.1.2.366 Uebung_077_AUI: RHS Position/Force/Draft auf UT ausgeben mit AR_Timeout_Zero (Adapter Version)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung gibt die Werte des ISOBUS-TECU-Heckkraftheber-Status (**RHS**, Rear Hitch Status) auf einem Universal Terminal (UT) aus: **Position**, **Force** (Nominal Lower Link Force) und **Draft**. Jeder Rohwert wird in einen REAL-Wert umgesetzt (unskaliert, NOT_AVAILABLE/ERROR_INDICATOR werden zu `0.0` gefiltert), durch eine eigene `AR_Timeout_Zero`-Instanz gegen Kommunikationsausfall abgesichert und über `Q_NumericValue_PHYSA` angezeigt. Die Übung ist analog zu `TECU_RHS.SUB` der Krauternter-Software.

Verwendete Funktionsbausteine (FBs)

- **RHS_Inst**: ISOBUS-Adapter für den Heckkraftheber-Status
  - **Typ**: `isobus::tecu::IA_RHS`
  - **Parameter**: `QI = TRUE`
  - **Erklärung**: Liefert die Adapter `POSITION` und `FORCE` (je AUS, USINT-Rohwert), `DRAFT` (AUI, UINT-Rohwert) sowie `TIMEOUT` (AX, TRUE bei Kommunikationsausfall).
- **AX_SPLIT_3_Timeout**: Verteiler für den Timeout-Status
  - **Typ**: `adapter::events::unidirectional::AX_SPLIT_3`
  - **Parameter**: keine
  - **Erklärung**: Verteilt `RHS_Inst.TIMEOUT` auf `OUT1` (⟶ `TZ_Position.G`), `OUT2` (⟶ `TZ_Force.G`) und `OUT3` (⟶ `TZ_Draft.G`).
- **Conv_Position**, **Conv_Force**: Umsetzung USINT ⟶ REAL
  - **Typ**: `MyLib::sys::AUS_FIELDBUS_USINT_TO_AR` (SubApp-Instanzen)
  - **Parameter**: keine
  - **Erklärung**: Wandeln den TECU-Rohwert (`IN`, AUS) in einen AR-Wert (`OUT`) um. NOT_AVAILABLE/ERROR_INDICATOR werden gefiltert (Ausgabe `0.0`), sonst bleibt der Wert unskaliert (`SCALE = 1.0`, `OFFSET = 0` fest).
- **Conv_Draft**: Umsetzung UINT ⟶ REAL
  - **Typ**: `MyLib::sys::AUI_FIELDBUS_UINT_TO_AR` (SubApp-Instanz)
  - **Parameter**: keine
  - **Erklärung**: Wie oben, jedoch für den UINT-Rohwert (`IN`, AUI).
- **TZ_Position**, **TZ_Force**, **TZ_Draft**: Timeout-Nullung (SubApp-Instanzen)

  - **Typ**: `MyLib::sys::AR_Timeout_Zero`
  - **Parameter**: keine
  - **Erklärung**: Liefern `IN0` unverändert an `OUT`, außer `G` ist TRUE (Timeout) - dann `0.0`.
- **Q_NumericValue_RHS_Position**, **Q_NumericValue_RHS_Force**, **Q_NumericValue_RHS_Draft**: Ausgabe auf VT-Zahlenfelder
  - **Typ**: `isobus::UT::Q::Q_NumericValue_PHYSA`
  - **Parameter**: `stObj` = `NumberVariable_TECU_RHS_Position_N` / `NumberVariable_TECU_RHS_Force_N` / `NumberVariable_TECU_RHS_Draft_N`
  - **Erklärung**: Schreiben den physikalischen Wert (`rPhys`) in das zugehörige VT-Objekt.

Programmablauf und Verbindungen

1. Position: `RHS_Inst.POSITION` ⟶ `Conv_Position.IN` ⟶ `TZ_Position.IN0`; `AX_SPLIT_3_Timeout.OUT1` ⟶ `TZ_Position.G`; `TZ_Position.OUT` ⟶ `Q_NumericValue_RHS_Position.rPhys`.
2. Force: `RHS_Inst.FORCE` ⟶ `Conv_Force.IN` ⟶ `TZ_Force.IN0`; `AX_SPLIT_3_Timeout.OUT2` ⟶ `TZ_Force.G`; `TZ_Force.OUT` ⟶ `Q_NumericValue_RHS_Force.rPhys`.
3. Draft: `RHS_Inst.DRAFT` ⟶ `Conv_Draft.IN` ⟶ `TZ_Draft.IN0`; `AX_SPLIT_3_Timeout.OUT3` ⟶ `TZ_Draft.G`; `TZ_Draft.OUT` ⟶ `Q_NumericValue_RHS_Draft.rPhys`.
4. `RHS_Inst.TIMEOUT` ⟶ `AX_SPLIT_3_Timeout.IN`: ein Timeout-Status steuert alle drei Nullungen.

Verhalten: Solange Daten eintreffen, zeigt das UT Position, Kraft und Zugkraft des Heckkrafthebers. Bei Kommunikationsausfall (`TIMEOUT` = TRUE) zeigen alle drei Felder `0.0`.

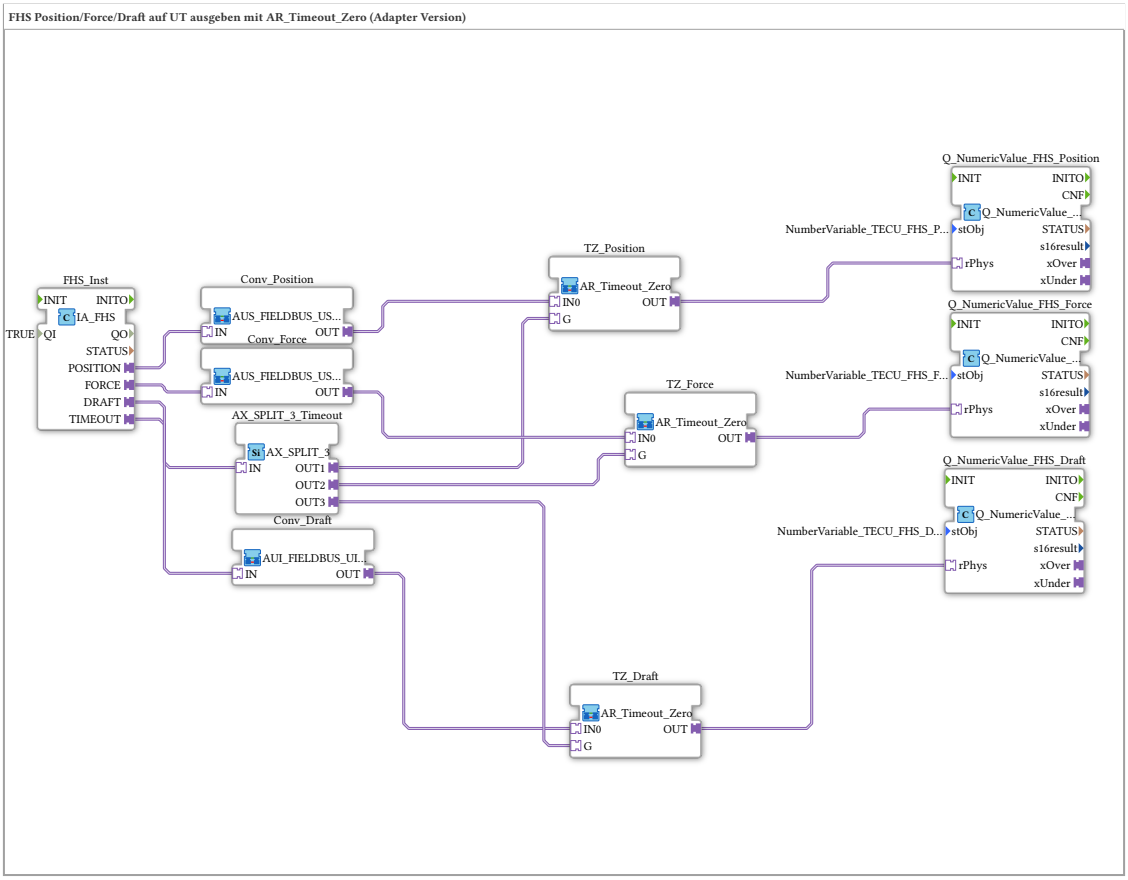
Zusammenfassung

Die Übung überträgt das Muster aus `Uebung_073d_AUI` und `Uebung_075a_AUI` auf drei Werte mit unterschiedlichen Rohdatentypen (USINT und UINT): jeweils eine passende Konverter-SubApp, eine `AR_Timeout_Zero`-Instanz und ein `Q_NumericValue_PHYSA`, wobei ein gemeinsamer `AX_SPLIT_3` den Timeout-Status verteilt.

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

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

2.1.2.367 Uebung_078_AUI: FHS Position/Force/Draft auf UT ausgeben mit AR_Timeout_Zero (Adapter Version)



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung gibt die drei Messwerte des Front Hitch Status (FHS) - Position, Kraft (Force) und Zugkraft (Draft) - auf einem User Terminal (UT) aus. Jeder Rohwert wird in einen REAL-Wert umgewandelt und durch einen `AR_Timeout_Zero`-Baustein geführt, der bei einem Timeout der FHS-Nachricht den Wert 0.0 liefert, statt den letzten Wert einzufrieren. Das gemeinsame Timeout-Signal wird dafür auf drei Ziele verteilt. Alle Verbindungen laufen über Adapter.

Verwendete Funktionsbausteine (FBs)

- **FHS_Inst**: ISOBUS-Adapter für den Front Hitch Status
  - **Typ**: `isobus::tecu::IA_FHS`
  - **Parameter**: `QI = TRUE`
  - **Erklärung**: Stellt `POSITION` und `FORCE` als AUS-Adapter, `DRAFT` als AUI-Adapter und den Timeout-Status `TIMEOUT` als AX-Adapter bereit (PGN 65094).
- **Conv_Position**, **Conv_Force**: Umwandlung AUS ⟶ AR
  - **Typ**: `MyLib::sys::AUS_FIELDBUS_USINT_TO_AR` (SubApp-Instanz, siehe AUS_FIELDBUS_USINT_TO_AR)
  - **Parameter**: keine
  - **Erklärung**: Wandelt den TECU-Rohwert (USINT) unskaliert (`SCALE` = 1.0, `OFFSET` = 0) in einen AR-Wert um. Bei `NOT_AVAILABLE` oder `ERROR_INDI` wird 0.0 geliefert.
- **Conv_Draft**: Umwandlung AUI ⟶ AR
  - **Typ**: `MyLib::sys::AUI_FIELDBUS_UINT_TO_AR` (SubApp-Instanz)
  - **Parameter**: keine
  - **Erklärung**: Wie oben, jedoch für den UINT-Rohwert des AUI-Adapters `DRAFT`.
- **TZ_Position**, **TZ_Force**, **TZ_Draft**: Timeout-Zero
  - **Typ**: `MyLib::sys::AR_Timeout_Zero` (SubApp-Instanz)

  - **Parameter**: keine
  - **Erklärung**: Fendt-Schaltung: Der Adapter `OUT` liefert den Live-Wert `IN0` unverändert, außer `G` (in der Regel `IA_*.TIMEOUT`) ist TRUE - dann 0.0 statt des sonst eingefrorenen letzten Werts. Intern besteht der Baustein aus einem `initval_AR` mit `INIT_VAL = 0.0` und einem `AR_AX_SEL_AR`.
- **AX_SPLIT_3_Timeout**: Verteiler des Timeout-Signals
  - **Typ**: `adapter::events::unidirectional::AX_SPLIT_3`
  - **Parameter**: keine
  - **Erklärung**: Verteilt das Timeout-Signal von `FHS_Inst.TIMEOUT` auf die drei Timeout-Zero-Bausteine.
- **Q_NumericValue_FHS_Position**, **Q_NumericValue_FHS_Force**, **Q_NumericValue_FHS_Draft**: VT-Anzeigen
  - **Typ**: `isobus::UT::Q::Q_NumericValue_PHYSA`
  - **Parameter**: `stObj = NumberVariable_TECU_FHS_Position_N`, `NumberVariable_TECU_FHS_Force_N` bzw. `NumberVariable_TECU_FHS_Draft_N`
  - **Erklärung**: Zeigen den physikalischen REAL-Wert vom Adapter `rPhys` auf dem im Objektpool-Eintrag `stObj` angegebenen VT-Objekt an.

Programmablauf und Verbindungen

1. Position: `FHS_Inst.POSITION` ⟶ `Conv_Position.IN` ⟶ `TZ_Position.IN0` (`Conv_Position.OUT`) ⟶ `Q_NumericValue_FHS_Position.rPhys` (`TZ_Position.OUT`).
2. Force: `FHS_Inst.FORCE` ⟶ `Conv_Force.IN` ⟶ `TZ_Force.IN0` (`Conv_Force.OUT`) ⟶ `Q_NumericValue_FHS_Force.rPhys` (`TZ_Force.OUT`).
3. Draft: `FHS_Inst.DRAFT` ⟶ `Conv_Draft.IN` ⟶ `TZ_Draft.IN0` (`Conv_Draft.OUT`) ⟶ `Q_NumericValue_FHS_Draft.rPhys` (`TZ_Draft.OUT`).
4. `FHS_Inst.TIMEOUT` ⟶ `AX_SPLIT_3_Timeout.IN`: Das Timeout-Signal wird verteilt.
5. `AX_SPLIT_3_Timeout.OUT1` ⟶ `TZ_Posistion.G`, `AX_SPLIT_3_Timeout.OUT2` ⟶ `TZ_Force.G`, `AX_SPLIT_3_Timeout.OUT3` ⟶ `TZ_Draft.G`: Bei Timeout liefern alle drei Timeout-Zero-Bausteine 0.0.

Verhalten: Im Normalbetrieb zeigt das UT die drei umgewandelten FHS-Werte; bei Timeout der FHS-Nachricht zeigen alle drei Felder 0.0.

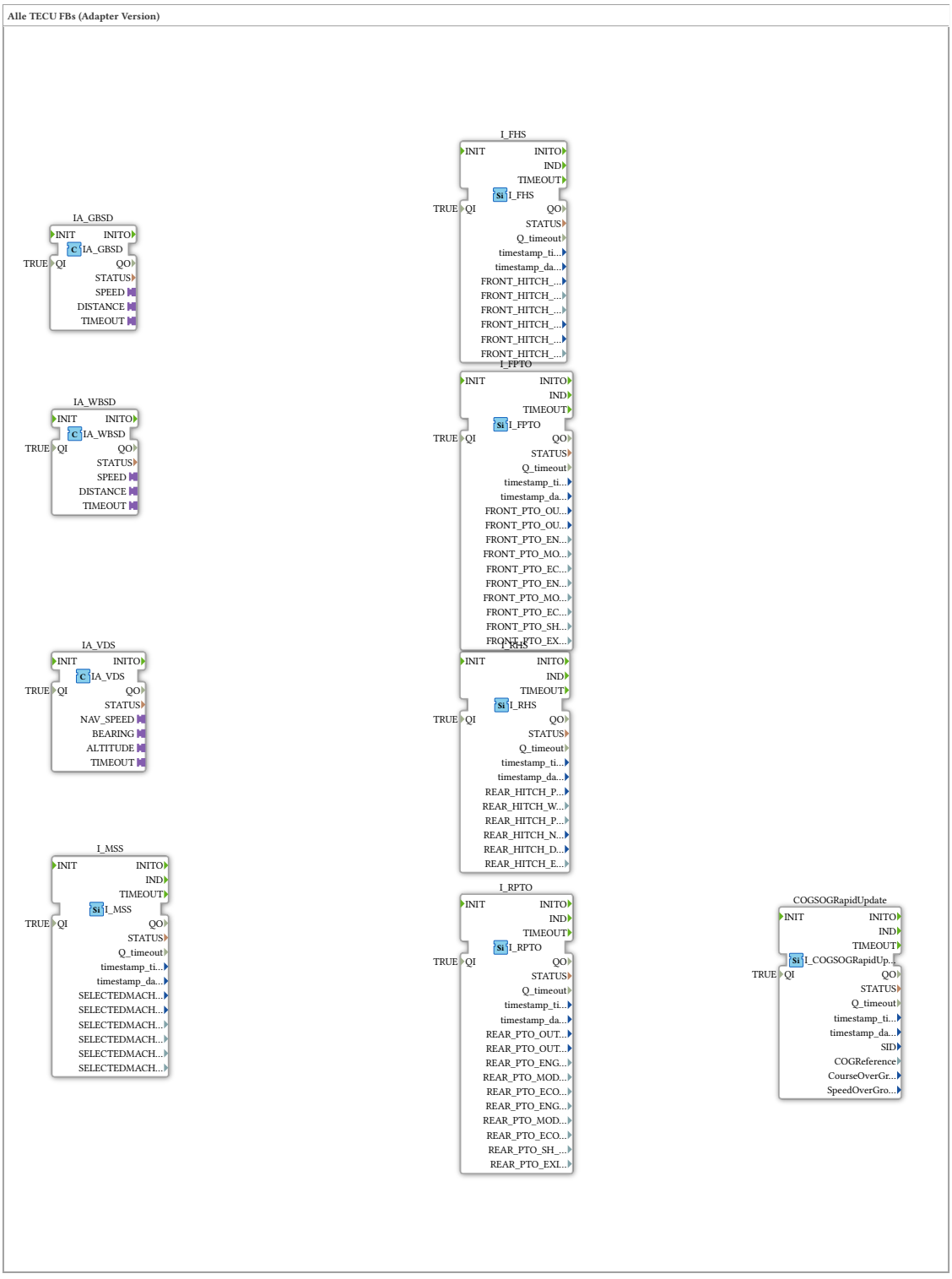
Zusammenfassung

Die Übung zeigt, wie sich die Fendt-Schaltung mit dem wiederverwendbaren Baustein `AR_Timeout_Zero` für mehrere Messwerte gleichzeitig einsetzen lässt: Ein einziges Timeout-Signal wird über `AX_SPLIT_3` auf drei Kanäle verteilt, sodass bei Kommunikationsausfall alle Anzeigen auf 0.0 fallen.

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

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

2.1.2.368 Uebung_079_AX: Alle TECU FBs (Adapter Version)



• — ○ • — ○ • — ○ •

Einleitung

Diese Übung dient der Einführung in die wichtigsten ISOBUS-konformen Funktionsbausteine (FBs) für die TECU-Plattform in der Adapter-Version. Sie lernen die grundlegenden Schnittstellenbausteine kennen, die in landwirtschaftlichen Anwendungen zur Steuerung und Überwachung von Traktor- oder Implementfunktionen eingesetzt werden. Die Übung stellt eine Sammlung aller benötigten ISOBUS-Adapter-FBs bereit, die als Grundlage für komplexere Steuerungsaufgaben innerhalb der 4diac-IDE dienen.

Verwendete Funktionsbausteine (FBs)

Die Übung enthält ausschließlich die folgenden vordefinierten ISOBUS-Adapter-Funktionsbausteine aus der Bibliothek `isobus::tecu`. Jeder Baustein besitzt einen booleschen Eingang `QI` (Quality/Enable), der zur Aktivierung auf `TRUE` gesetzt ist. Es werden keine zusätzlichen Verbindungen zwischen den FBs hergestellt.

- **IA_GBSD** – Getriebe-/Bremssteuerung (Generic Brake System Device)
- **IA_VDS** – Virtual Display Server (Anzeige und Bedienung)
- **IA_WBSD** – Working Body Set Device (Arbeitsgeräte)
- **I_MSS** – Management System Server (Systemverwaltung)
- **I_FHS** – Front Hitch System (Frontkraftheber)
- **I_FPTO** – Front Power Take-Off (Frontzapfwelle)
- **I_RHS** – Rear Hitch System (Heckkraftheber)
- **I_RPTO** – Rear Power Take-Off (Heckzapfwelle)
- **COGSOGRapidUpdate** – (Course Guidance Update, Spurführung)

Alle Bausteine sind vom Typ `isobus::tecu::<Name>` und werden im Netzwerk ohne weitere Verschaltung nur zur Bereitstellung der Schnittstellen verwendet.

Programmablauf und Verbindungen

In dieser Übung werden die ISOBUS-Adapter-FBs **nicht** miteinander verbunden oder in einen Ablauf integriert. Ziel ist es, die einzelnen Bausteine und ihre jeweiligen Funktionsbereiche kennenzulernen. Die Bausteine sind als reine Adapter-Versionen ausgeführt und können später in eigenen Anwendungen als Verbindungspunkte zu realen TECU-Geräten oder Simulationen eingesetzt werden.

Mögliche Lernziele:

- Erkennen der ISOBUS-Adapter-Struktur (QI-Eingang, Ereignis-, Datenports)
- Verstehen der Aufgabenbereiche der gängigen TECU-Funktionen (Zapfwelle, Kraftheber, Anzeige etc.)
- Vorbereitung für die Zusammenschaltung mehrerer Adapter zu einer funktionalen Steuerung

Die Übung kann direkt nach dem Import in die 4diac-IDE gestartet werden. Es sind keine weiteren Vorkenntnisse außer der grundlegenden Bedienung der IDE erforderlich.

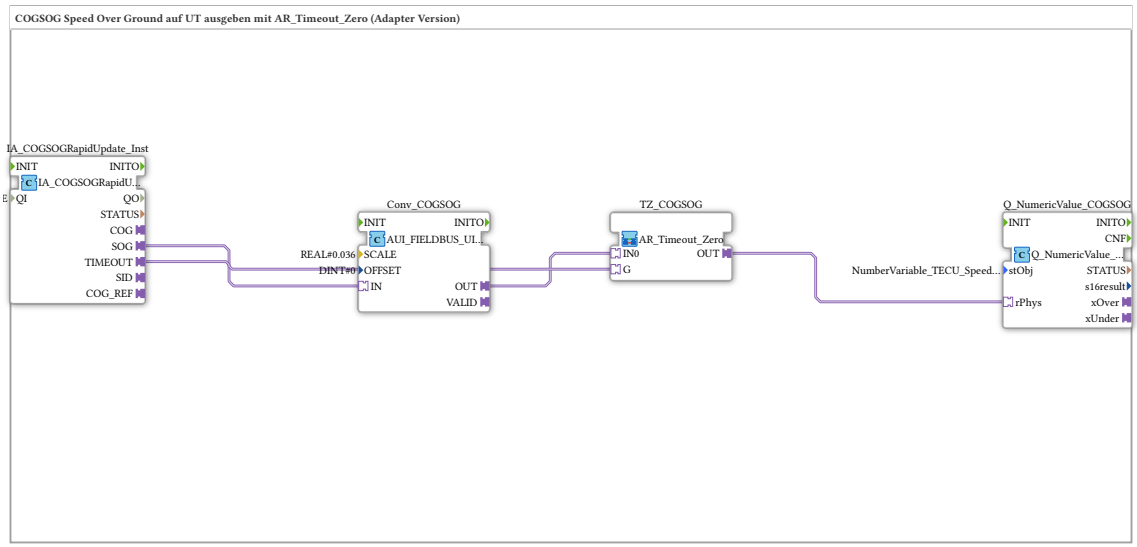
Zusammenfassung

Die Übung Uebung_079_AX stellt eine vollständige Auflistung aller wesentlichen ISOBUS-Adapter-FBs für TECU zur Verfügung. Sie bietet einen idealen Einstieg in die Modellierung landwirtschaftlicher Steuerungen mit 4diac und bildet die Basis für weiterführende Übungen, in denen die Bausteine miteinander verbunden und mit eigenen Logiken ergänzt werden.

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

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

2.1.2.369 Uebung_079a_AUI: COGSOG Speed Over Ground auf UT ausgeben mit AR_Timeout_Zero (Adapter-Version)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung gibt die Geschwindigkeit über Grund (COGSOG Rapid Update, Speed Over Ground) des TECU auf dem ISOBUS-Terminal aus: mit PHYS-Skalierung (`0.036`) und der Fendt-Schaltung `AR_Timeout_Zero` (Adapter-Version, analog zur Krauternter-SubApp `TECU_COGSOG.SUB`). Bei einem Kommunikationsausfall wird `0.0` angezeigt.

Verwendete Funktionsbausteine (FBs)

- **IA_COGSOGRapidUpdate_Inst**: ISOBUS-Adapter für Course Over Ground und Speed Over Ground (Rapid Update)
  - **Typ**: `isobus::tecu::IA_COGSOGRapidUpdate`
  - **Parameter**: `QI = TRUE`
  - **Erklärung**: Liefert unter anderem die Geschwindigkeit über Grund als `AUI`-Adapter `SOG` und den Timeout-Status als `AX`-Adapter `TIMEOUT`.
- **Conv_COGSOG**: Skalierung des Feldbuswerts
  - **Typ**: `logiBUS::signalprocessing::fieldbus::AUI_FIELDBUS_UINT_TO_SIGNAL_SCALED`
  - **Parameter**: `SCALE = 0.036`, `OFFSET = 0`
  - **Erklärung**: Wandelt den `AUI`-Wert (UINT vom Feldbus) nur bei gültigem Signal in einen skalierten `AR`-Wert (REAL) um; `OFFSET` wird nach der Skalierung addiert.
- **TZ_COGSOG**: Fendt-Schaltung (SubApp)
  - **Typ**: `MyLib::sys::AR_Timeout_Zero`
  - **Parameter**: keine
  - **Erklärung**: Gibt `IN0` unverändert aus, außer `G` (hier `IA_COGSOGRapidUpdate_Inst.TIMEOUT`, echter Kommunikationsausfall) ist `TRUE`: Dann wird `0.0` statt des sonst eingefrorenen letzten Werts ausgegeben.
- **Q_NumericValue_COGSOG**: Anzeige auf dem ISOBUS-Terminal
  - **Typ**: `isobus::UT::Q::Q_NumericValue_PHYSA`
  - **Parameter**: `stObj = NumberVariable_TECU_Speed_COGSOG_N`
  - **Erklärung**: Zeigt den physikalischen Wert (`rPhys`) im Objekt `NumberVariable_TECU_Speed_COGSOG_N` an.

Programmablauf und Verbindungen

1. `IA_COGSOGRapidUpdate_Inst.SOG` ⟶ `Conv_COGSOG.IN` (AdapterConnection): Der Geschwindigkeitswert über Grund wird skaliert.
2. `Conv_COGSOG.OUT` ⟶ `TZ_COGSOG.IN0` (AdapterConnection): Der skalierte Wert ist der Live-Wert der Fendt-Schaltung.

3. `IA_COGSOGRapidUpdate_Inst.TIMEOUT` ⟶ `TZ_COGSOG.G` (AdapterConnection): Der Timeout-Status steuert, ob `0.0` ausgegeben wird.
4. `TZ_COGSOG.OUT` ⟶ `Q_NumericValue_COGSOG.rPhys` (AdapterConnection): Das Ergebnis wird auf dem Terminal angezeigt.

Verhalten: Solange der TECU die Botschaft liefert, zeigt das Terminal den mit `0.036` skalierten Wert. Fällt die Kommunikation aus (`TIMEOUT`), zeigt es `0.0`.

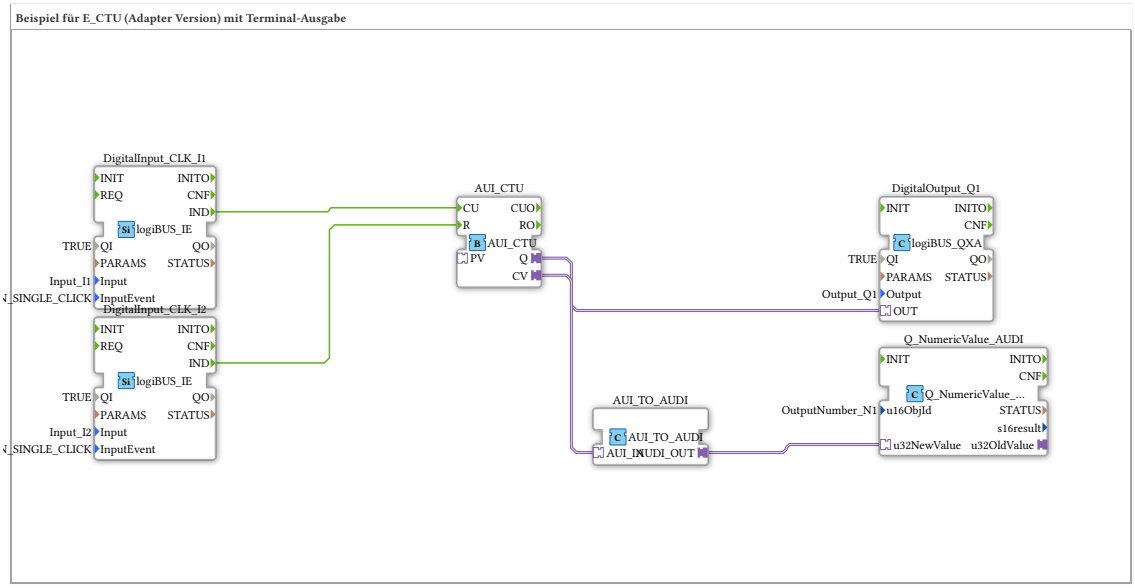
Zusammenfassung

Die Übung entspricht im Aufbau `Uebung_070d_AUI` (WBSD), verwendet aber den Adapter `IA_COGSOGRapidUpdate` und den Skalierungsfaktor `0.036` für die Geschwindigkeit über Grund.

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

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

2.1.2.370 Uebung_080_AUI: Beispiel für E_CTU (Adapter Version) mit Terminal-Ausgabe



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert die Anwendung eines **Adapter-basierten Aufwärtszählers (AUI_CTU)** in 4diac. Der Zähler wird über einen Ereigniseingang von einem Taster (Input_I1) inkrementiert und über einen zweiten Taster (Input_I2) zurückgesetzt. Der aktuelle Zählwert wird sowohl auf einem digitalen Ausgang (Output_Q1) als auch als numerischer Wert über einen Terminal-Ausgang ausgegeben. Die Übung vermittelt den Umgang mit der Adapter-Schnittstelle für Ereignisse und deren Konvertierung in Datenwerte.

Verwendete Funktionsbausteine (FBs)

Der Netzwerk-Editor der Subapplikation enthält sechs Funktionsbausteine. Nachfolgend werden diese im Einzelnen beschrieben.

DigitalInput_CLK_I1

- **Typ**: `logiBUS::io::DI::logiBUS_IE`
- **Parameter**:
  - `QI` = `TRUE`
  - `Input` = `Input_I1`
  - `InputEvent` = `BUTTON_SINGLE_CLICK`
- **Funktion**: Dieser Baustein reagiert auf einen einzelnen Tastendruck am physikalischen Eingang `Input_I1` und erzeugt das Ereignis `IND` am Ausgang.

DigitalInput_CLK_I2

- **Typ**: `logiBUS::io::DI::logiBUS_IE`
- **Parameter**:
  - `QI` = `TRUE`
  - `Input` = `Input_I2`
  - `InputEvent` = `BUTTON_SINGLE_CLICK`
- **Funktion**: Analog zu `DigitalInput_CLK_I1`, jedoch für den Reset-Taster an `Input_I2`.

AUI_CTU

- **Typ**: `adapter::events::unidirectional::AUI_CTU`
- **Parameter**: Keine konfigurierten Parameter im XML.
- **Funktion**: Dies ist ein Adapter-Funktionsbaustein, der einen Aufwärtszähler (Counter Up) realisiert. Er verfügt über die Ereigniseingänge `CU` (Inkrement) und `R` (Reset) sowie die Ereignisausgänge `Q` (Zählerstand erreicht) und `CV` (aktueller Zählwert als Adapter-Ausgang). Die Zählschwelle (PV) ist standardmäßig auf einen vorgegebenen Wert gesetzt.

AUI_TO_AUDI

- **Typ**: `adapter::conversion::unidirectional::AUI_TO_AUDI`
- **Parameter**: Keine konfigurierten Parameter.
- **Funktion**: Dieser Baustein wandelt einen AUI-Adapterausgang (Ereignis mit Wert) in einen reinen Datenwert (AUDI) um. Er erhält am Adaptereingang `AUI_IN` das Signal `CV` und liefert am Datenausgang `AUDI_OUT` den aktuellen Zählwert als `UINT`-Wert.

Q_NumericValue_AUDI

- **Typ**: `isobus::UT::Q::Q_NumericValue_AUDI`
- **Parameter**:
  - `u16ObjId` = `OutputNumber_N1`
- **Funktion**: Dieser Baustein empfängt einen numerischen Wert (über den Adaptereingang `u32NewValue`) und stellt ihn als Ausgabe auf dem Terminal dar. Der Parameter `u16ObjId` legt die Objektkennung für die Terminalausgabe fest.

DigitalOutput_Q1

- **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
- **Parameter**:
  - `QI` = `TRUE`
  - `Output` = `Output_Q1`
- **Funktion**: Dieser Baustein setzt den physikalischen Ausgang `Output_Q1` auf `TRUE`, sobald am Ereigniseingang `OUT` ein Ereignis eintrifft. Er dient dazu, den Zählerstand (z.B. Erreichen einer Schwelle) als binäres Signal anzuzeigen.

Programmablauf und Verbindungen

Die Übung ist als Subapplikation (`Uebung_080_AUI`) angelegt und benötigt keine eigenen Schnittstellen – alle Ein- und Ausgänge sind interne Hardwarezuordnungen.

Ereignisverbindungen:

- Das Ereignis `IND` von `DigitalInput_CLK_I1` wird an den Ereigniseingang `CU` des AUI_CTU angeschlossen. Jeder Tastendruck an Input_I1 erhöht den Zähler um 1.
- Das Ereignis `IND` von `DigitalInput_CLK_I2` wird an den Ereigniseingang `R` des AUI_CTU angeschlossen. Ein Tastendruck an Input_I2 setzt den Zähler zurück.

Adapterverbindungen:

- Der Adapterausgang `Q` von `AUI_CTU` (zeigt an, dass der Zählerstand die Schwelle erreicht hat) ist mit dem Ereigniseingang `OUT` von `DigitalOutput_Q1` verbunden. Bei Erreichen der Schwelle wird der Ausgang `Output_Q1` aktiviert.
- Der Adapterausgang `CV` von `AUI_CTU` (aktueller Zählwert) ist mit dem Adapteingang `AUI_IN` des Konverters `AUI_TO_AUDI` verbunden.
- Der Datenausgang `AUDI_OUT` von `AUI_TO_AUDI` liefert den Zählwert als Ganzzahl und wird mit dem Adaptereingang `u32NewValue` des Terminal-Bausteins `Q_NumericValue_AUDI` verbunden. Dadurch wird der aktuelle Zählerstand kontinuierlich auf dem Terminal ausgegeben.

Ablauf:

1. Nach dem Start der Applikation ist der Zählerstand 0.
2. Jeder Druck auf `Input_I1` erhöht den Zähler um
3. Der neue Wert wird sofort auf dem Terminal angezeigt.
4. Wird der voreingestellte Schwellwert (PV) erreicht, wird `Output_Q1` auf `TRUE` gesetzt.
5. Ein Druck auf `Input_I2` setzt den Zähler zurück auf 0 (auch der Ausgang wird wieder `FALSE`).

Zusammenfassung

Die Übung zeigt, wie ein Adapter-basierter Zähler (AUI_CTU) in 4diac mit Hardware-Eingängen und -Ausgängen verknüpft wird. Durch die Verwendung des Konverters `AUI_TO_AUDI` wird der adapternative Wert in einen einfachen Datenwert umgewandelt, der anschließend auf einem Terminal ausgegeben werden kann. Die separate Ansteuerung von Zähleingang und Reset sowie die binäre Rückmeldung über einen digitalen Ausgang machen diese Übung zu einem grundlegenden Beispiel für zeit- und ereignisgesteuerte Zählfunktionen in der IEC 61499-Architektur.

Lernziele:

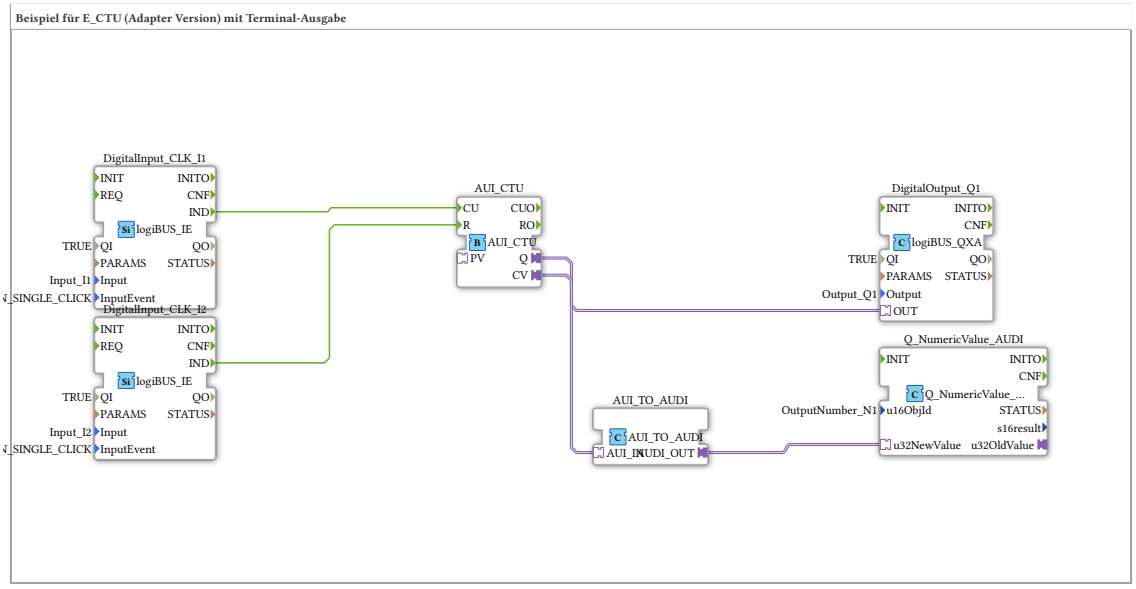
- Verständnis der Adapter-Schnittstellen (AUI) für Ereignisse und Daten.
- Einbindung von Hardware-Eingängen (Taster) und -Ausgängen in ein Steuerungsprogramm.
- Konvertierung zwischen Adapter- und Datenformaten.
- Nutzung eines Terminal-Ausgabe-FBs zur Laufzeitbeobachtung.

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

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

2.1.2.371 Uebung_080_AX: Beispiel für E_CTU (Adapter Version) mit Terminal-Ausgabe

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_080_AX (Beispiel für E_CTU (Adapter Version) mit Terminal-Ausgabe).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Beispiel für E_CTU (Adapter Version) mit Terminal-Ausgabe**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_080_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
- **DigitalInput_CLK_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **AUI_CTU**: Instanz des Typs adapter::events::unidirectional::AUI_CTU.
- **DigitalInput_CLK_I2**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I2
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **AUI_TO_AUDI**: Instanz des Typs adapter::conversion::unidirectional::AUI_TO_AUDI.
- **Q_NumericValue_AUDI**: Instanz des Typs isobus::UT::Q::Q_NumericValue_AUDI.
  - Parameter u16ObjId = OutputNumber_N1

Verbindungen und Schnittstellen

Adapterverbindungen:

- AUI_CTU.Q -> DigitalOutput_Q1.OUT
- AUI_CTU.CV -> AUI_TO_AUDI.AUI_IN
- AUI_TO_AUDI.AUDI_OUT -> Q_NumericValue_AUDI.u32NewValue

Ereignisverbindungen:

- DigitalInput_CLK_I1.IND -> AUI_CTU.CU
- DigitalInput_CLK_I2.IND -> AUI_CTU.R

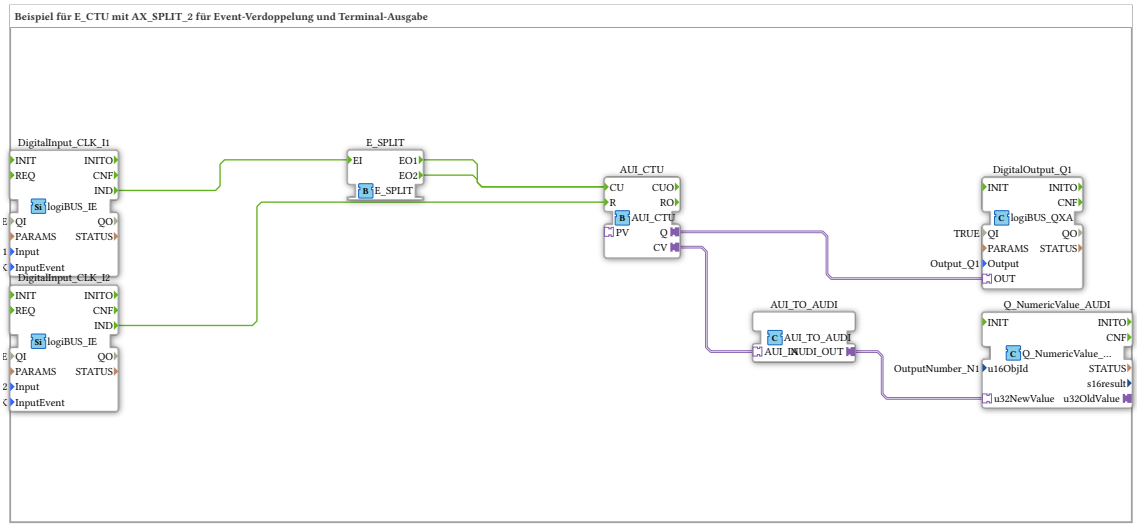
Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_080_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.372 Uebung_080b_AUI: Beispiel für E_CTU mit AX_SPLIT_2 für Event-Verdoppelung und Terminal-Ausgabe



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung demonstriert den Einsatz eines aufwärtszählenden Zählers (E_CTU) mit Ereignisverdoppelung durch einen E_SPLIT-Funktionsbaustein. Zwei Hardware-Taster (an Input_I1 und Input_I2) dienen als Zählimpulsgeber und Rücksetzsignal. Der aktuelle Zählerstand wird über einen Adapter-Ausgang (CV) ausgegeben und auf einem numerischen Terminal (OutputNumber_N1) dargestellt. Ein zusätzlicher digitaler Ausgang (Output_Q1) signalisiert den Q-Zustand des Zählers.

Lernziele:

- Anwendung des E_CTU-Zählers (Aufwärtszähler)
- Nutzung des E_SPLIT zur Verdoppelung eines Ereignissignals
- Verwendung von AUI-Adapter-Schnittstellen für Daten- und Ereignisübertragung
- Parametrierung von logiBUS-Eingängen und -Ausgängen
- Ausgabe numerischer Werte auf einem Terminal (Q_NumericValue_AUDI)

Schwierigkeitsgrad: Mittel **Vorkenntnisse**: Grundlagen der 4diac-IDE, Umgang mit Ereignis- und Datenverbindungen, Ein/Ausgabe-Konfiguration

Verwendete Funktionsbausteine (FBs)

Folgende Funktionsbausteine werden im SubApp-Netzwerk eingesetzt:

`DigitalInput_CLK_I1` (Typ: `logiBUS::io::DI::logiBUS_IE`)

- **Parameter**: `QI=TRUE`, `Input=Input_I1`, `InputEvent=BUTTON_SINGLE_CLICK`
- **Funktion**: Liest einen digitalen Eingang (Taster I1) und generiert bei einem einfachen Klick das Ereignis `IND`.

`DigitalInput_CLK_I2` (Typ: `logiBUS::io::DI::logiBUS_IE`)

- **Parameter**: `QI=TRUE`, `Input=Input_I2`, `InputEvent=BUTTON_SINGLE_CLICK`
- **Funktion**: Liest einen digitalen Eingang (Taster I2) und generiert bei einem einfachen Klick das Ereignis `IND` (dient als Rücksetzsignal).

`E_SPLIT` (Typ: `iec61499::events::E_SPLIT`)

- **Parameter**: keine
- **Funktion**: Teilt ein eingehendes Ereignis (EI) auf zwei identische Ausgangsereignisse (EO1, EO2) auf.

`E_CTU` (Typ: `adapter::events::unidirectional::AUI_CTU`)

- **Parameter**: keine
- **Funktion**: Aufwärtszähler mit zwei Ereigniseingängen: CU (Count Up) und R (Reset). Über den Adapter-Ausgang `Q` wird der Zählerstand als boolescher Wert (bei CV>0) ausgegeben, über `CV` der aktuelle Zählerstand (Datenadapter).

`AUI_TO_AUDI` (Typ: `adapter::conversion::unidirectional::AUI_TO_AUDI`)

- **Parameter**: keine
- **Funktion**: Wandelt einen AUI-Datenadapter (hier den Zählerwert CV) in einen AUDI-Datenadapter (u32) um, der von nachfolgenden Bausteinen verarbeitet werden kann.

`DigitalOutput_Q1` (Typ: `logiBUS::io::DQ::logiBUS_QXA`)

- **Parameter**: `QI=TRUE`, `Output=Output_Q1`
- **Funktion**: Steuert einen digitalen Ausgang (Q1) basierend auf dem eingehenden Wert am Adapteranschluss `OUT`.

`Q_NumericValue_AUDI` (Typ: `isobus::UT::Q::Q_NumericValue_AUDI`)

- **Parameter**: `u16ObjId=OutputNumber_N1`
- **Funktion**: Nimmt einen 32-Bit-Zahlenwert (via `u32NewValue`) entgegen und gibt diesen auf dem konfigurierten Terminalobjekt (hier `OutputNumber_N1`) aus.

Programmablauf und Verbindungen

1. **Ereigniserzeugung**:
   - Bei Betätigung des Tasters an `Input_I1` erzeugt `DigitalInput_CLK_I1` ein Ereignis `IND`.

- Bei Betätigung des Tasters an `Input_I2` erzeugt `DigitalInput_CLK_I2` ein Ereignis `IND`.

2. **Ereignisverdoppelung**:
   - Das `IND`-Ereignis von I1 wird an den Eingang `EI` von `E_SPLIT` geleitet.
   - `E_SPLIT` gibt zwei identische Ereignisse an seinen Ausgängen `EO1` und `EO2` aus.
   - Beide Ereignisse werden – über separate Verbindungen – an den CU-Eingang des `E_CTU` angeschlossen. **Dadurch wird jeder Tastendruck auf I1 als zwei Zählimpulse gewertet.**
3. **Zähler**:
   - Jedes CU-Ereignis erhöht den internen Zähler von `E_CTU` um 1.
   - Das `IND`-Ereignis von I2 (Rücksetztaster) ist mit dem Eingang `R` von `E_CTU` verbunden und setzt den Zähler auf 0 zurück.
4. **Ausgabe**:
   - Der Zählerausgang `Q` (Adapter) wird mit dem Adaptereingang `OUT` von `DigitalOutput_Q1` verbunden. Solange der Zählerstand > 0 ist, wird der digitale Ausgang Q1 aktiv (TRUE).
   - Der Zählerstand `CV` (ebenfalls ein AUI-Adapter) wird über `AUI_TO_AUDI` in einen AUDI-Adapter konvertiert und an `Q_NumericValue_AUDI.u32NewValue` übergeben. Der Baustein zeigt den aktuellen Zählerwert auf dem konfigurierten Terminal (OutputNumber_N1) an.

Zusammenfassung der Verbindungen (aus dem XML):

Quelle	Ziel	Typ
`DigitalInput_CLK_I1.IND`	`E_SPLIT.EI`	Event
`E_SPLIT.EO1`	`E_CTU.CU`	Event
`E_SPLIT.EO2`	`E_CTU.CU`	Event
`DigitalInput_CLK_I2.IND`	`E_CTU.R`	Event
`E_CTU.Q`	`DigitalOutput_Q1.OUT`	Adapter
`E_CTU.CV`	`AUI_TO_AUDI.AUI_IN`	Adapter
`AUI_TO_AUDI.AUDI_OUT`	`Q_NumericValue_AUDI.u32NewValue`	Adapter

Zusammenfassung

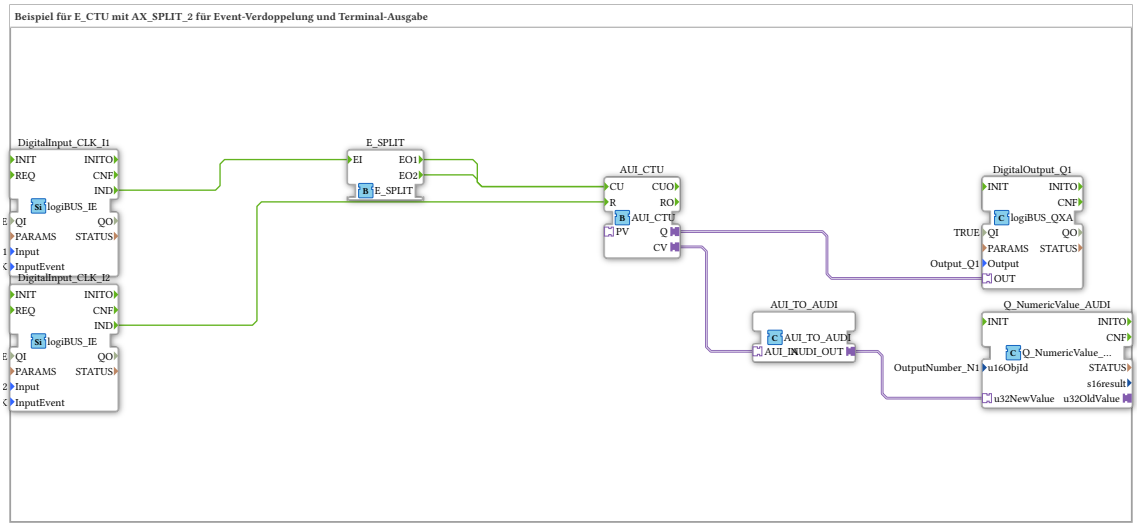
Die Übung **Uebung_080b_AUI** veranschaulicht die Kombination eines Aufwärtszählers mit einer Ereignisverdoppelung durch `E_SPLIT`. Der Zähler reagiert auf zwei Taster: einer zählt (mit verdoppelter Impulsrate), der andere setzt zurück. Die Ergebnisse werden sowohl auf einem digitalen Ausgang (Q1) als auch auf einem Terminal (OutputNumber_N1) ausgegeben. Sie lernen dabei den Umgang mit Adapter-basierten Daten- und Ereignisschnittstellen sowie die Konfiguration von logiBUS-Ein- und -Ausgängen in der 4diac-IDE.

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

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

2.1.2.373 Uebung_080b_AX: Beispiel für E_CTU mit AX_SPLIT_2 für Event-Verdoppelung und Terminal-Ausgabe

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_080b_AX (Beispiel für E_CTU mit AX_SPLIT_2 für Event-Verdoppelung und Terminal-Ausgabe).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Beispiel für E_CTU mit AX_SPLIT_2 für Event-Verdoppelung und Terminal-Ausgabe**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_080b_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
- **DigitalInput_CLK_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **AUI_CTU**: Instanz des Typs adapter::events::unidirectional::AUI_CTU.
- **DigitalInput_CLK_I2**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I2
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **E_SPLIT**: Instanz des Typs iec61499::events::E_SPLIT.
- **AUI_TO_AUDI**: Instanz des Typs adapter::conversion::unidirectional::AUI_TO_AUDI.
- **Q_NumericValue_AUDI**: Instanz des Typs isobus::UT::Q::Q_NumericValue_AUDI.
  - Parameter u16ObjId = OutputNumber_N1

Verbindungen und Schnittstellen

Adapterverbindungen:

- AUI_CTU.Q -> DigitalOutput_Q1.OUT
- AUI_CTU.CV -> AUI_TO_AUDI.AUI_IN
- AUI_TO_AUDI.AUDI_OUT -> Q_NumericValue_AUDI.u32NewValue

Ereignisverbindungen:

- DigitalInput_CLK_I1.IND -> E_SPLIT.EI
- E_SPLIT.EO1 -> AUI_CTU.CU
- E_SPLIT.EO2 -> AUI_CTU.CU
- DigitalInput_CLK_I2.IND -> AUI_CTU.R

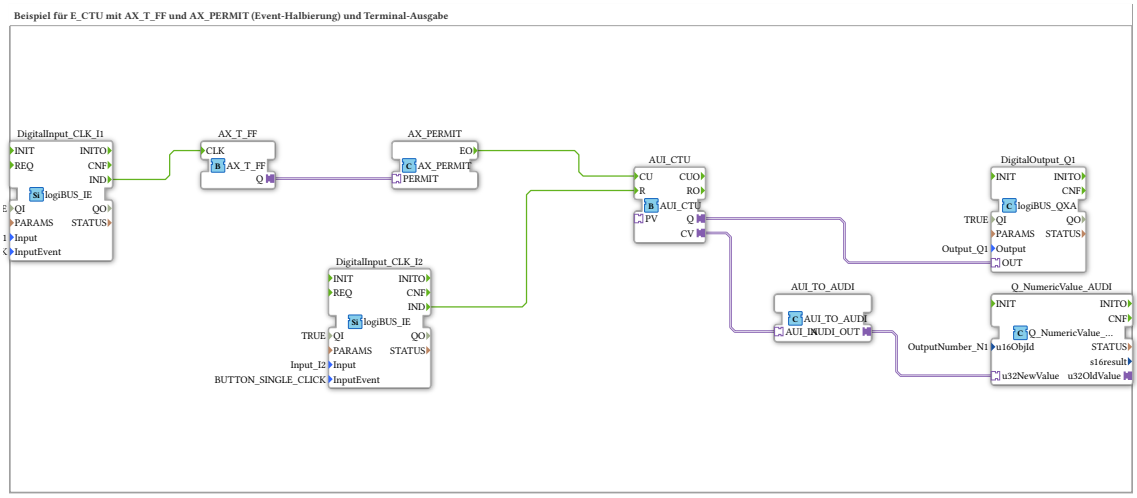
Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_080b_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.374 Uebung_080c_AUI: Beispiel für E_CTU mit AX_T_FF und AX_PERMIT (Event-Halbierung) und Terminal-Ausgabe



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert die Verwendung eines Aufwärtszählers (E_CTU) in Kombination mit einem T-Flipflop (AX_T_FF) und einem Ereignis-Freigabeblock (AX_PERMIT). Die Ereignisse eines Tasters (I1) werden durch das T-Flipflop halbiert, bevor sie den Zähler erhöhen. Ein zweiter Taster (I2) dient als Reset. Der aktuelle Zählerstand wird über einen Konverter an eine numerische Ausgabe auf einem Terminal gesendet, und der Zählerstatus (Q) schaltet einen digitalen Ausgang.

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_CLK_I1** (Typ: `logiBUS::io::DI::logiBUS_IE`)
  - Parameter: QI = TRUE, Input = `Input_I1`, InputEvent = `BUTTON_SINGLE_CLICK`
  - Ereignisausgang: `IND` ⟶ verbunden mit `AX_T_FF.CLK`
- **DigitalInput_CLK_I2** (Typ: `logiBUS::io::DI::logiBUS_IE`)
  - Parameter: QI = TRUE, Input = `Input_I2`, InputEvent = `BUTTON_SINGLE_CLICK`
  - Ereignisausgang: `IND` ⟶ verbunden mit `E_CTU.R`
- **AX_T_FF** (Typ: `adapter::events::unidirectional::AX_T_FF`)
  - Adaptereingang: `CLK` ⟵ `DigitalInput_CLK_I1.IND`
  - Adapterausgang: `Q` ⟶ `AX_PERMIT.PERMIT`
- **AX_PERMIT** (Typ: `adapter::events::unidirectional::AX_PERMIT`)
  - Adaptereingang: `PERMIT` ⟵ `AX_T_FF.Q`
  - Ereignisausgang: `EO` ⟶ `E_CTU.CU`
- **E_CTU** (Typ: `adapter::events::unidirectional::AUI_CTU`)
  - Ereigniseingang: `CU` ⟵ `AX_PERMIT.EO`

  - Ereigniseingang: `R` ⟵ `DigitalInput_CLK_I2.IND`
  - Adapterausgang: `Q` ⟶ `DigitalOutput_Q1.OUT`
  - Adapterausgang: `CV` ⟶ `AUI_TO_AUDI.AUI_IN`
- **DigitalOutput_Q1** (Typ: `logiBUS::io::DQ::logiBUS_QXA`)
  - Parameter: QI = TRUE, Output = `Output_Q1`
  - Adaptereingang: `OUT` ⟵ `E_CTU.Q`
- **AUI_TO_AUDI** (Typ: `adapter::conversion::unidirectional::AUI_TO_AUDI`)
  - Adaptereingang: `AUI_IN` ⟵ `E_CTU.CV`
  - Adapterausgang: `AUDI_OUT` ⟶ `Q_NumericValue_AUDI.u32NewValue`
- **Q_NumericValue_AUDI** (Typ: `isobus::UT::Q::Q_NumericValue_AUDI`)
  - Parameter: `u16ObjId` = `OutputNumber_N1`
  - Dateneingang: `u32NewValue` ⟵ `AUI_TO_AUDI.AUDI_OUT`

Programmablauf und Verbindungen

1. **Ereignisquelle I1** Ein Taster an `Input_I1` (konfiguriert auf `BUTTON_SINGLE_CLICK`) löst bei jedem Drücken ein einzelnes Ereignis am Ausgang `IND` von `DigitalInput_CLK_I1` aus.
2. **Ereignishalbierung** Das Ereignis wird an den Takteingang `CLK` des T-Flipflops `AX_T_FF` weitergeleitet. Dieses wechselt bei jedem Takt seinen Zustand (Q) und gibt somit nur jedes zweite Tasterereignis als aktiven Zustand weiter.
3. **Freigabe durch AX_PERMIT** Der Ausgang `Q` von `AX_T_FF` wird mit dem `PERMIT`-Eingang von `AX_PERMIT` verbunden. Solange `PERMIT` aktiv ist, wird ein eingehendes Ereignis am internen Eingang (hier nicht sichtbar) zum Ausgang `EO` durchgeschaltet. Dadurch wird die Ereignisfrequenz halbiert.
4. **Zähler E_CTU** Das freigegebene Ereignis erreicht den Aufwärtszähler über seinen Eingang `CU`. Bei jedem Ereignis wird der interne Zählwert um 1 erhöht. Ein zweiter Taster an `Input_I2` löst ein Ereignis aus, das über `DigitalInput_CLK_I2.IND` direkt mit dem Reset-Eingang `R` von `E_CTU` verbunden ist – bei Betätigung wird der Zähler auf 0 zurückgesetzt.
5. **Ausgabe des Zählerstands**
   - Der aktuelle Zählerwert (`CV`) wird über den Konverter `AUI_TO_AUDI` in einen analogen Datenwert umgewandelt.
   - Dieser Wert wird an den Datenbaustein `Q_NumericValue_AUDI` übergeben, der ihn auf einem Terminal (z. B. über den konfigurierten `OutputNumber_N1`) anzeigt.
   - Gleichzeitig liefert der Zähler einen binären Ausgang `Q`, der immer dann aktiv ist, wenn der Zählerstand > 0 ist. Dieser wird an den digitalen Ausgang `DigitalOutput_Q1` (an `Output_Q1`) weitergeleitet.

Lernziele & Voraussetzungen:

- Grundlegendes Verständnis von IEC 61499-Ereignissteuerung
- Umgang mit Zählern, Flipflops und Ereignis-Freigabe
- Einfache Terminalausgabe über numerische IDs

Start der Übung: Nach dem Laden der Subapplikation in die 4diac-IDE und dem Verbinden mit einer passenden Hardwareplattform (z. B. logiBUS) kann die Übung durch Drücken der Taster I1 und I2 gestartet werden.

Zusammenfassung

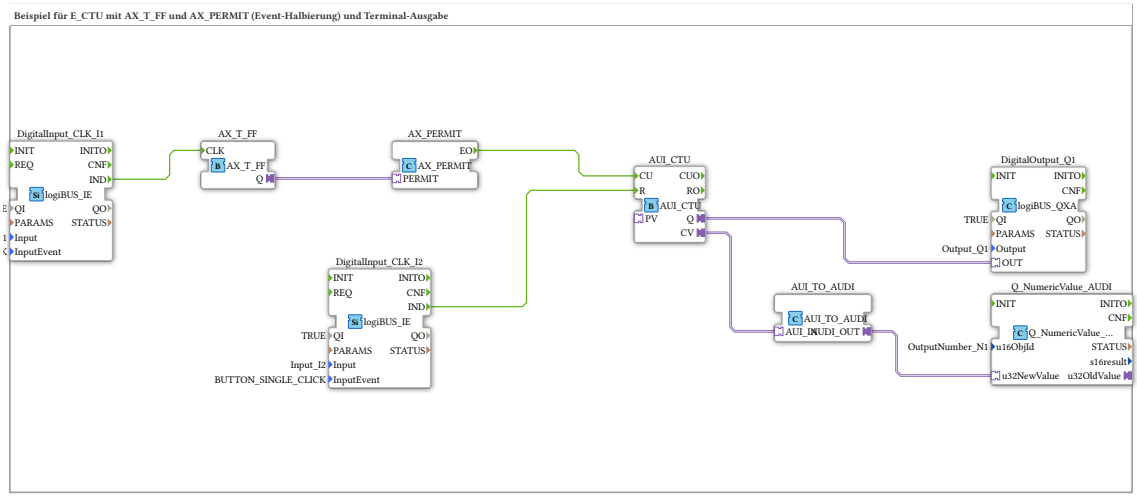
Die Übung `Uebung_080c_AUI` veranschaulicht die Kombination eines T-Flipflops mit einer Ereignisfreigabe zur Reduzierung der Ereignisfrequenz sowie die Verwendung eines Aufwärtszählers. Durch die Kopplung mit einem Terminal-Ausgabebaustein wird der Zählerstand visualisiert. Das Zusammenspiel von Ereignis-, Adapter- und Datenverbindungen zeigt typische Muster für modulare Automatisierungslösungen nach IEC 61499.

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

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

2.1.2.375 Uebung_080c_AX: Beispiel für E_CTU mit AX_T_FF und AX_PERMIT (Event-Halbierung) und Terminal-Ausgabe

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_080c_AX (Beispiel für E_CTU mit AX_T_FF und AX_PERMIT (Event-Halbierung) und Terminal-Ausgabe).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Beispiel für E_CTU mit AX_T_FF und AX_PERMIT (Event-Halbierung) und Terminal-Ausgabe**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_080c_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
- **DigitalInput_CLK_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **AUI_CTU**: Instanz des Typs adapter::events::unidirectional::AUI_CTU.
- **DigitalInput_CLK_I2**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I2
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **AX_PERMIT**: Instanz des Typs adapter::events::unidirectional::AX_PERMIT.
- **AX_T_FF**: Instanz des Typs adapter::events::unidirectional::AX_T_FF.
- **AUI_TO_AUDI**: Instanz des Typs adapter::conversion::unidirectional::AUI_TO_AUDI.
- **Q_NumericValue_AUDI**: Instanz des Typs isobus::UT::Q::Q_NumericValue_AUDI.
  - Parameter u16ObjId = OutputNumber_N1

Verbindungen und Schnittstellen

Adapterverbindungen:

- AX_T_FF.Q -> AX_PERMIT.PERMIT
- AUI_CTU.Q -> DigitalOutput_Q1.OUT
- AUI_CTU.CV -> AUI_TO_AUDI.AUI_IN
- AUI_TO_AUDI.AUDI_OUT -> Q_NumericValue_AUDI.u32NewValue

Ereignisverbindungen:

- DigitalInput_CLK_I1.IND -> AX_T_FF.CLK

- AX_PERMIT.EO -> AUI_CTU.CU
- DigitalInput_CLK_I2.IND -> AUI_CTU.R

Programmablauf und Funktionsweise

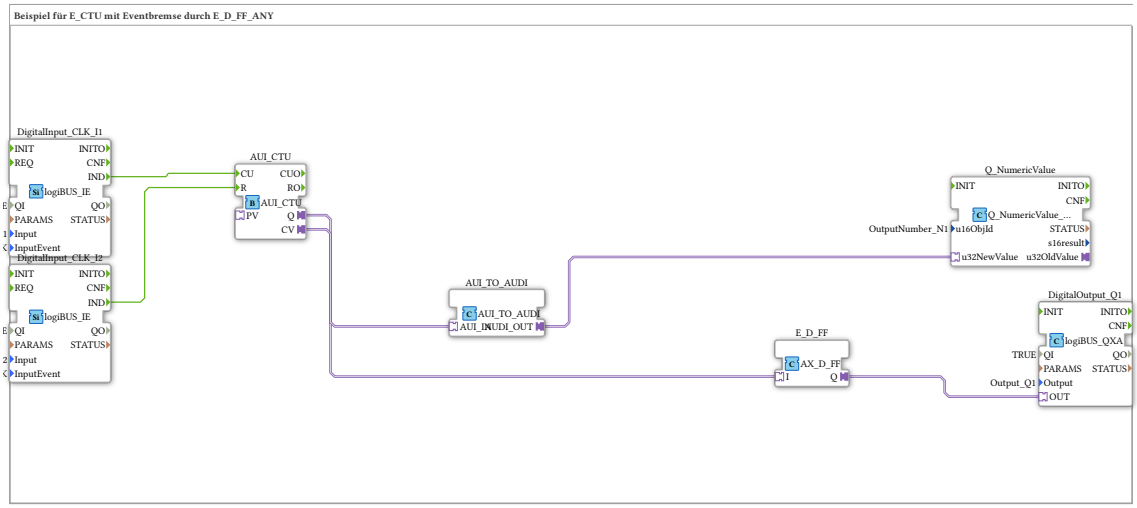
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_080c_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.376 Uebung_080d_AX: Beispiel für E_CTU mit Eventbremse durch E_D_FF_ANY

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_080d_AX (Beispiel für E_CTU mit Eventbremse durch E_D_FF_ANY).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Beispiel für E_CTU mit Eventbremse durch E_D_FF_ANY**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_080d_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
- **DigitalInput_CLK_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **AUI_CTU**: Instanz des Typs adapter::events::unidirectional::AUI_CTU.
- **DigitalInput_CLK_I2**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I2
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **E_D_FF**: Instanz des Typs adapter::events::unidirectional::AX_D_FF.

- **AUI_TO_AUDI**: Instanz des Typs adapter::conversion::unidirectional::AUI_TO_AUDI.
- **Q_NumericValue**: Instanz des Typs isobus::UT::Q::Q_NumericValue_AUDI.
  - Parameter u16ObjId = OutputNumber_N1

Verbindungen und Schnittstellen

Adapterverbindungen:

- AUI_CTU.Q -> E_D_FF.I
- E_D_FF.Q -> DigitalOutput_Q1.OUT
- AUI_CTU.CV -> AUI_TO_AUDI.AUI_IN
- AUI_TO_AUDI.AUDI_OUT -> Q_NumericValue.u32NewValue

Ereignisverbindungen:

- DigitalInput_CLK_I1.IND -> AUI_CTU.CU
- DigitalInput_CLK_I2.IND -> AUI_CTU.R

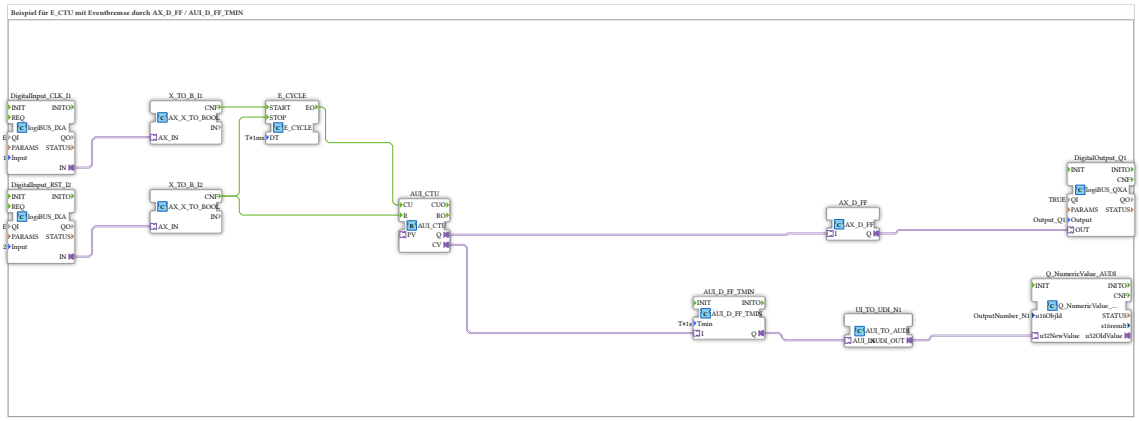
Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_080d_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.377 Uebung_080e2_AX: Beispiel für E_CTU mit Eventbremse durch AX_D_FF / AUI_D_FF_TMIN



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung zeigt den Einsatz eines ereignisgesteuerten Zählers (E_CTU) in Kombination mit einer **Eventbremse**, die mit den Funktionsbausteinen `AX_D_FF` und `AUI_D_FF_TMIN` realisiert wird. Der Zähler wird über einen zyklischen Ereignisgeber (`E_CYCLE`) gesteuert, der durch einen externen digitalen Eingang (CLK) freigegeben wird. Ein zweiter digitaler Eingang (RST) setzt den Zähler zurück und stoppt den Zyklus. Der Zählerstand wird über eine minimale Haltezeit (`Tmin = 1 s`) auf einen numerischen Ausgang `N1` übertragen. Zusätzlich wird ein digitaler Ausgang `Q1` bei Erreichen des Zählerendwerts aktiviert.

Verwendete Funktionsbausteine (FBs)

Die Übung besteht aus einer Sub-Applikation, die folgende Bausteine enthält:

- **DigitalInput_CLK_I1** – `logiBUS_IXA` Digitaler Eingang `Input_I1` (CLK) als Quelle für die Freigabe des Zyklus.

- **DigitalInput_RST_I2** – `logiBUS_IXA` Digitaler Eingang `Input_I2` (RST) zum Zurücksetzen des Zählers und Stoppen des Zyklus.
- **X_TO_B_I1** – `AX_X_TO_BOOL` Konvertiert den Adapterwert von `DigitalInput_CLK_I1` in einen booleschen Wert.
- **X_TO_B_I2** – `AX_X_TO_BOOL` Konvertiert den Adapterwert von `DigitalInput_RST_I2` in einen booleschen Wert.
- **E_CYCLE** – `E_CYCLE` Erzeugt periodisch alle 1 ms ein Ereignis (`EO`). Wird über `START`/`STOP` gesteuert.
- **E_CTU** – `AUI_CTU` Ereignisgesteuerter Vorwärtszähler. Der Zähleingang `CU` erhöht den internen Zählerstand `CV`. Bei Erreichen des maximalen Werts wird der Ausgang `Q` gesetzt. Der Eingang `R` setzt `CV` zurück.
- **AX_D_FF** – `AX_D_FF` D-Flipflop auf Ereignisbasis. Speichert ein Ereignis (über `I`) und gibt es am Ausgang `Q` weiter. Dient als Puffer für das Zählersignal `Q`.
- **AUI_D_FF_TMIN** – `AUI_D_FF_TMIN` D-Flipflop mit einstellbarer Mindestzeit (`Tmin = 1 s`). Ein anliegendes Ereignis wird erst nach Ablauf der Mindestzeit am Ausgang `Q` weitergegeben. Dient als **Eventbremse** für den Zählerstand `CV`.
- **UI_TO_UDI_N1** – `AUI_TO_AUDI` Konvertiert das ereignisbasierte Signal des Flipflops `AUI_D_FF_TMIN` in ein datenorientiertes Signal.
- **Q_NumericValue** – `Q_NumericValue_AUDI` Schreibt den konvertierten Zählerstand auf den numerischen Ausgang `OutputNumber_N1`.
- **DigitalOutput_Q1** – `logiBUS_QXA` Digitaler Ausgang `Output_Q1`, gesteuert durch das Flipflop `AX_D_FF`.

Programmablauf und Verbindungen

Der Ablauf wird über Ereignis- und Adapterverbindungen gesteuert:

1. **Eingangsverarbeitung**
   - `DigitalInput_CLK_I1` liefert den Taktgeber (CLK) über `X_TO_B_I1` an den `E_CYCLE.START` (Ereignis CNF).
   - `DigitalInput_RST_I2` liefert den Reset (RST) über `X_TO_B_I2` an `E_CTU.R` und gleichzeitig an `E_CYCLE.STOP`.
2. **Zyklischer Zähler**
   - `E_CYCLE` erzeugt alle 1 ms ein Ereignis `EO`, sobald `START` aktiv ist. Dieses wird an `E_CTU.CU` geleitet. Der Zähler erhöht seinen Stand `CV` bei jedem Ereignis.
   - Bei Aktivierung des Reset-Eingangs (RST) wird der Zähler zurückgesetzt und der Zyklus gestoppt.
3. **Ausgabe des Zählerstands**
   - Der aktuelle Zählerstand `CV` von `E_CTU` wird über den Adapter an `AUI_D_FF_TMIN.I` übergeben.
   - `AUI_D_FF_TMIN` gibt diesen Wert nach einer Verzögerung von mindestens 1 s als Ereignis an `AUI_D_FF_TMIN.Q` weiter (Eventbremse).
   - Über `UI_TO_UDI_N1` (Konvertierung) gelangt der Wert an `Q_NumericValue` und wird auf den numerischen Ausgang `N1` ausgegeben.
4. **Digitaler Ausgang**
   - Wenn der Zähler seinen Endwert erreicht hat, setzt `E_CTU.Q` ein Ereignis. Dieses wird vom `AX_D_FF` gespeichert und an den Digitalausgang `DigitalOutput_Q1` weitergegeben.
   - Der Ausgang bleibt solange gesetzt, bis ein neues Ereignis (z. B. durch Reset) den Zustand ändert.

- **Lernziele**: Verständnis für ereignisgesteuerte Zähler, Steuerung von Zyklen mit externen Eingängen, Einsatz von Flipflops als Entprellung oder Mindesthaltezeit (Eventbremse).
- **Schwierigkeitsgrad**: Fortgeschritten
- **Vorkenntnisse**: Grundlagen der Ereignisverarbeitung in 4diac, Umgang mit Ein-/Ausgangsbausteinen, Konzepte von Zählern und Flipflops.
- **Hinweis**: Die Übung kann direkt in einer 4diac-IDE geladen und ausgeführt werden, sofern die benötigten Bibliotheken (logiBUS, isobus, adapter) verfügbar sind.

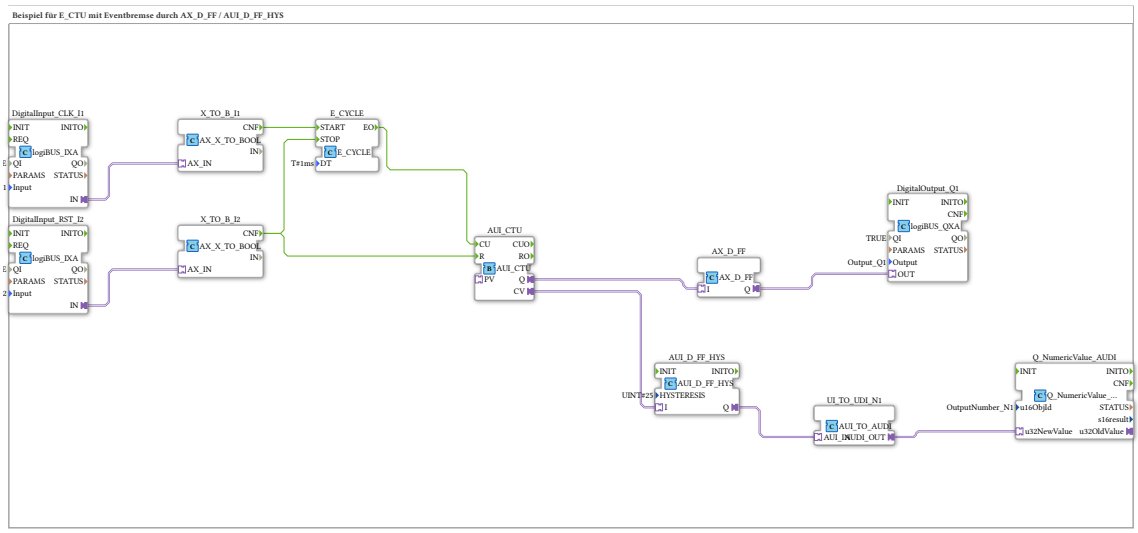
Zusammenfassung

Die Übung `Uebung_080e2_AX` demonstriert, wie ein ereignisgesteuerter Zähler (`E_CTU`) mit einer **Eventbremse** aus zwei unterschiedlichen Flipflops (`AX_D_FF`, `AUI_D_FF_TMIN`) kombiniert wird. Durch die Mindestzeit des Flipflops wird eine ungewollt schnelle Aktualisierung des Ausgangswerts verhindert. Der Zähler wird von einem zyklischen Ereignisgeber getaktet, der über einen externen Eingang freigegeben und zurückgesetzt werden kann. Das Zusammenspiel der Bausteine vermittelt wichtige Konzepte der ereignisbasierten Steuerung in der Automatisierungstechnik.

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

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

2.1.2.378 Uebung_080e3_AX: Beispiel für E_CTU mit Eventbremse durch AX_D_FF / AUI_D_FF_HYS



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert den Einsatz des universellen Aufwärtszählers **E_CTU** in Kombination mit einer **Eventbremse**, realisiert durch die Funktionsbausteine **AX_D_FF** (D-Flipflop) und **AUI_D_FF_HYS** (D-Flipflop mit Hysterese). Ziel ist es, den Zählerstand zu stabilisieren und auszugeben, sowie einen digitalen Ausgang bei Erreichen eines Zählerüberlaufs zu setzen.

Verwendete Funktionsbausteine (FBs)

Die Übung verwendet ausschließlich primitive Funktionsbausteine (keine Sub-Applikationen). Nachfolgend sind alle eingesetzten Bausteine mit ihren Parametern und Verbindungen aufgeführt.

Bausteinname	Typ	Parameter	Kurzbeschreibung
DigitalInput_CLK_I1	logiBUS_IXA	QI = TRUE, Input = Input_I1	Digitaler Eingang für das Taktsignal (CLK)
DigitalInput_RST_I2	logiBUS_IXA	QI = TRUE, Input = Input_I2	Digitaler Eingang für den Rücksetzimpuls (RST)
X_TO_B_I1	AX_X_TO_BOOL	–	Konvertiert den Adapter-Eingang (AX) in einen Booleschen Wert
X_TO_B_I2	AX_X_TO_BOOL	–	Gleiche Funktion für den Rücksetz-Eingang

E_CYCLE	E_CYCLE	DT = T#1ms	Zyklischer Taktgeber (Periode 1 ms)
E_CTU	AUI_CTU	–	Universeller Aufwärtszähler (Count Up)
AX_D_FF	AX_D_FF	–	D-Flipflop (AX-Welt), speichert einen booleschen Zustand
AUI_D_FF_HYS	AUI_D_FF_HYS	HYSTERESIS = UINT#25	D-Flipflop mit Hysterese auf dem Zählerstand
UI_TO_UDI_N1	AUI_TO_AUDI	–	Konvertiert AUI (unsigned integer) nach AUDI (adapter universal data interface)
Q_NumericValue	Q_NumericValue_AUDI	u16ObjId = OutputNumber_N1	Ausgabe eines numerischen Wertes über den Feldbus
DigitalOutput_Q1	logiBUS_QXA	QI = TRUE, Output = Output_Q1	Digitaler Ausgang

Programmablauf und Verbindungen

Der Ablauf lässt sich in mehrere Schritte gliedern:

1. **Eingangssignale erfassen** Die beiden digitalen Eingänge `DigitalInput_CLK_I1` und `DigitalInput_RST_I2` wandeln die physikalischen Signale in Adapterdaten um. Die nachgeschalteten Konverter `X_TO_B_I1` und `X_TO_B_I2` stellen daraus Boolesche Werte bereit (Ereignisausgang `CNF`).
2. **Taktgenerierung starten/stoppen**
   - Das Taktsignal `CLK_I1` (über `X_TO_B_I1.CNF`) startet den zyklischen Taktgeber `E_CYCLE` (`START`-Ereignis).
   - Das Rücksetzsignal `RST_I2` (über `X_TO_B_I2.CNF`) beendet den Taktgeber (`STOP`-Ereignis) und setzt gleichzeitig den Zähler `E_CTU` zurück (`R`-Ereignis).
3. **Zähler inkrementieren** Der Taktgeber erzeugt alle 1 ms ein `EO`-Ereignis, das den Zähler `E_CTU` am Eingang `CU` (Count Up) erhöht.
4. **Ausgabe des Zählerstands**
   - Der aktuelle Zählerstand (`CV`) wird dem Baustein `AUI_D_FF_HYS` übergeben. Dieses D-Flipflop mit Hysterese (Hysteresewert = 25) stabilisiert den Wert und gibt ihn an den Konverter `UI_TO_UDI_N1` weiter.
   - Der konvertierte Wert gelangt schließlich an `Q_NumericValue` und wird als numerischer Ausgang zur Verfügung gestellt.
5. **Überlauf signalisieren** Wenn der Zähler seinen Maximalwert erreicht (Überlauf, Ereignisausgang `Q`), wird das D-Flipflop `AX_D_FF` gesetzt. Dessen Ausgang `Q` schaltet den digitalen Ausgang `DigitalOutput_Q1` ein.

Verbindungsübersicht (grafisch) *Das Bildmaterial kann aus der 4diac-IDE exportiert werden.*

Zusammenfassung

Die Übung veranschaulicht die Kopplung eines zyklischen Taktgebers mit einem Aufwärtszähler, einer **Ereignisbremse** (zur Vermeidung von schnellen Zustandswechseln) sowie einer **Hysterese** zur Glättung des Zählerstands. Nach Abschluss dieser Übung sind Sie in der Lage:

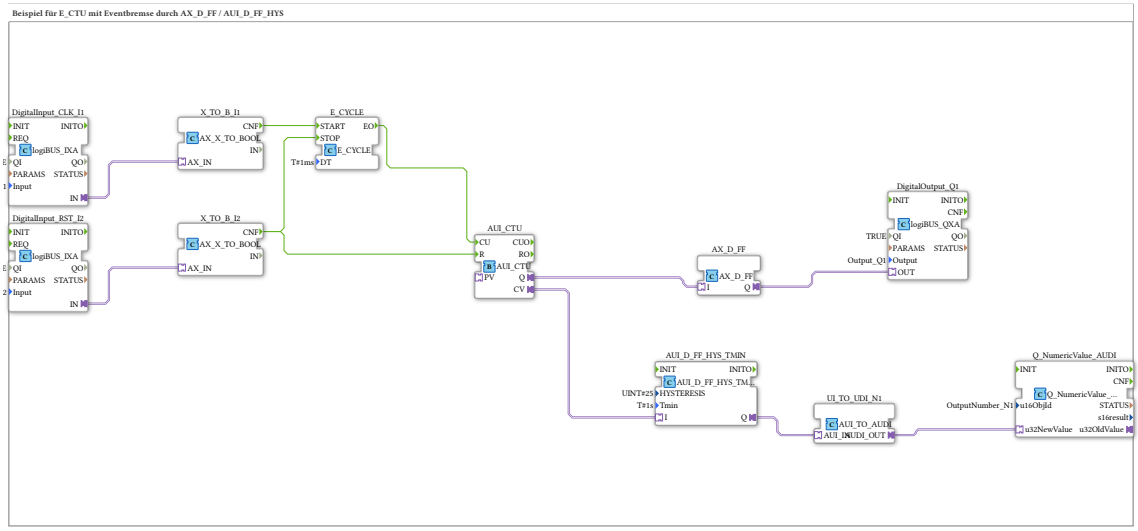
- den Baustein `E_CTU` zu parametrieren und in eine Steuerungslogik einzubinden,
- D-Flipflops zur Zustandsspeicherung zu verwenden,
- eine Hysterese zur Stabilisierung von Zählerwerten anzuwenden,
- und die Verbindung von Adapter-Bausteinen in der 4diac-IDE nachzuvollziehen.

Lernziele: Ereignisgesteuerte Zähler, Zustandsautomaten, Hysteresefilter. **Voraussetzungen**: Grundlegende Bedienung der 4diac-IDE, Verständnis von Ereignis- und Datenflüssen. **Schwierigkeitsgrad**: Mittel

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

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

2.1.2.379 Uebung_080e4_AX: Beispiel für E_CTU mit Eventbremse durch AX_D_FF / AUI_D_FF_HYS



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung zeigt die Verwendung eines **E_CTU**-Zählers in Kombination mit einer Ereignisbremse, die durch die Funktionsbausteine **AX_D_FF** (D-Flipflop) und **AUI_D_FF_HYS** (D-Flipflop mit Hysterese und Mindestzeit) realisiert wird. Der Zählerwert wird über einen digitalen Ausgang (Q1) und numerisch über ein Bus-Objekt (N1) ausgegeben. Die Schaltung dämpft sprunghafte Änderungen des Zählerstandes und ermöglicht eine gezielte Filterung von Ereignissen.

Verwendete Funktionsbausteine (FBs)

Die Subapplikation enthält keine eigenen Unterbausteine. Sie setzt sich aus vordefinierten Funktionsbausteinen zusammen:

- **DigitalInput_CLK_I1** – `logiBUS::io::DI::logiBUS_IXA` Liest den binären Eingang **I1** (Taktgeber). Wird als Startimpuls für den Ereigniszyklus verwendet.
- **DigitalInput_RST_I2** – `logiBUS::io::DI::logiBUS_IXA` Liest den binären Eingang **I2** (Reset). Setzt den Zähler zurück und stoppt den Ereigniszyklus.
- **X_TO_B_I1** – `adapter::conversion::unidirectional::AX_X_TO_BOOL` Wandelt den AX-Wert von `DigitalInput_CLK_I1` in ein BOOL-Signal.
- **X_TO_B_I2** – `adapter::conversion::unidirectional::AX_X_TO_BOOL` Wandelt den AX-Wert von `DigitalInput_RST_I2` in ein BOOL-Signal.
- **E_CYCLE** – `iec61499::events::E_CYCLE` Erzeugt zyklisch ein Ereignis (Periode: 1 ms). Wird durch den Eingang I1 gestartet und durch I2 gestoppt.
- **E_CTU** – `adapter::events::unidirectional::AUI_CTU` Aufwärtszähler. Zählt jedes Ereignis an seinem **CU**-Eingang. Der aktuelle Zählwert liegt an **CV**, der Überlauf-/Grenzwert-Ausgang an **Q**. Der Zähler wird durch den Eingang **R** zurückgesetzt.

- **AX_D_FF** – `adapter::events::unidirectional::AX_D_FF` D-Flipflop. Übernimmt den logischen Zustand an **I** und gibt ihn über **Q** aus. Dient als Ereignisbremse: Der Zählerausgang Q des E_CTU wird nur nach dem Flipflop an den Digitalausgang weitergegeben.
- **AUI_D_FF_HYS** – `adapter::events::unidirectional::AUI_D_FF_HYS_TMIN` D-Flipflop mit Hysterese und Mindestzeit. Parameter:
  - `HYSTERESIS = UINT#25` – Hysteresebreite
  - `Tmin = T#1s` – minimale Verweildauer vor Zustandswechsel Filtert den Zählerwert **CV**, sodass nur signifikante Änderungen (größer als die Hysterese) nach Ablauf der Mindestzeit weitergegeben werden.
- **UI_TO_UDI_N1** – `adapter::conversion::unidirectional::AUI_TO_AUDI` Wandelt den gefilterten Zählerwert vom AUI-Format in ein AUDI-Format um.
- **Q_NumericValue** – `isobus::UT::Q::Q_NumericValue_AUDI` Gibt den numerischen Wert über das Bus-Objekt **OutputNumber_N1** aus.
- **DigitalOutput_Q1** – `logiBUS::io::DQ::logiBUS_QXA` Setzt den digitalen Ausgang **Q1** entsprechend dem Zustand des Flipflops `AX_D_FF`.

Programmablauf und Verbindungen

1. **Start/Stop der Zyklusabtastung** Solange der Eingang **I1** (Taktgeber) aktiv ist, läuft der Ereigniszyklus `E_CYCLE` mit einer Periode von 1 ms. Wird **I2** (Reset) aktiv, wird der Zyklus gestoppt und gleichzeitig der Zähler zurückgesetzt.
2. **Zählerbetrieb** Jedes vom `E_CYCLE` erzeugte Ereignis (**EO**) wird an den **CU**-Eingang des `E_CTU` weitergeleitet – der Zähler erhöht sich bei jedem Zyklus um 1, solange kein Reset erfolgt.
3. **Ereignisbremse mit D-Flipflop** Der Überlauf-/Grenzwert-Ausgang **Q** des Zählers wird auf das **I** des `AX_D_FF` gegeben. Das Flipflop übernimmt diesen Zustand ereignisgesteuert und gibt ihn an den Digitalausgang `DigitalOutput_Q1` weiter. Dadurch wird der Ausgang nicht zyklisch durchgeschaltet, sondern nur bei einer Änderung des Zähler-Q aktualisiert.
4. **Hysterese und Mindestzeit (Filterung des Zählerwerts)** Der aktuelle Zählerwert **CV** wird dem Baustein `AUI_D_FF_HYS` zugeführt. Dieser gibt den Wert nur dann an den numerischen Ausgang weiter, wenn die Änderung größer als die eingestellte Hysterese (25 Einheiten) ist und der neue Wert mindestens 1 Sekunde stabil bleibt. So werden kurzzeitige Sprünge im Zählerstand unterdrückt.
5. **Ausgabe** Der gefilterte Wert wird über `UI_TO_UDI_N1` und `Q_NumericValue` als numerisches Signal auf **OutputNumber_N1** ausgegeben. Der Digitalausgang **Q1** zeigt den Zustand des D-Flipflops an.

Lernziele:

- Verständnis des Zusammenspiels von Ereigniszyklen (`E_CYCLE`) und Zählern (`E_CTU`)
- Einsatz von D-Flipflops als Ereignisbremse
- Filterung von Zählerwerten mittels Hysterese und Mindestzeit
- Umgang mit Adapter-Bausteinen und Formatwandlungen (AX ↔ BOOL, AUI ↔ AUDI)

Schwierigkeitsgrad: Fortgeschritten **Benötigte Vorkenntnisse:** Grundlegende 4diac-IDE Bedienung, Wissen über Ereignis- und Datenflüsse, einfache Zähler und Flipflops.

Zusammenfassung

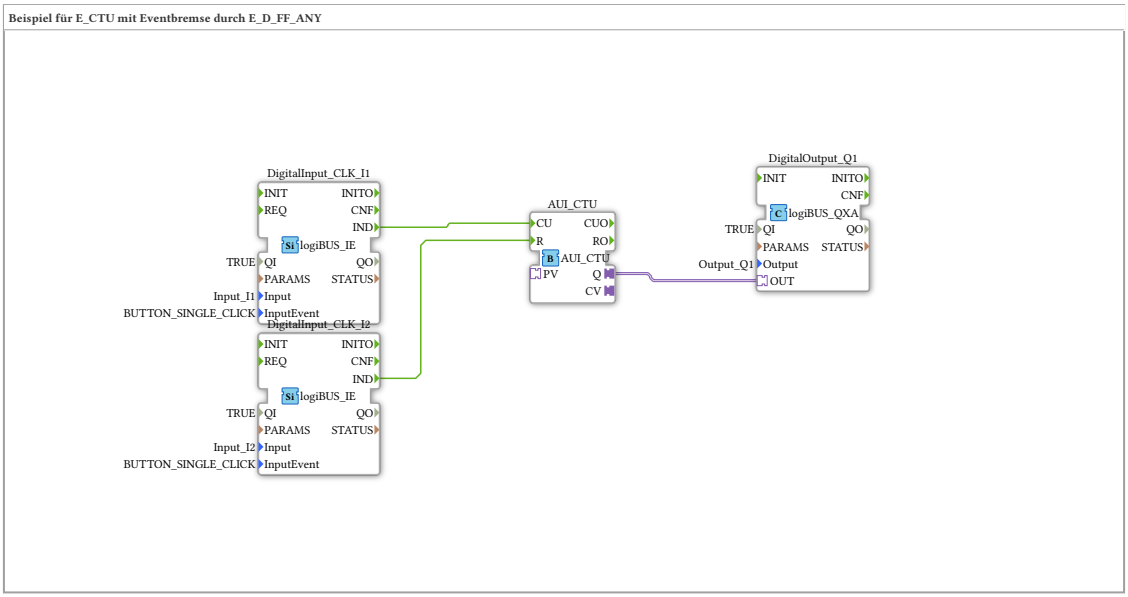
Die Übung `Uebung_080e4_AX` demonstriert einen Aufwärtszähler, dessen Ausgang durch ein D-Flipflop (Ereignisbremse) entkoppelt und dessen Zählwert durch einen weiteren D-Flipflop mit Hysterese und Mindestzeit gefiltert wird. Dadurch wird ein stabiler, geglätteter Zählerstand sowohl digital als auch numerisch ausgegeben. Die Schaltung eignet sich besonders für Anwendungen, in denen kurze Störungen oder schnelle Zählereignisse unterdrückt werden müssen.

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

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

2.1.2.380 Uebung_080e_AX: Beispiel für E_CTU mit Eventbremse durch E_D_FF_ANY

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_080e_AX (Beispiel für E_CTU mit Eventbremse durch E_D_FF_ANY).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Beispiel für E_CTU mit Eventbremse durch E_D_FF_ANY**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_080e_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
- **DigitalInput_CLK_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **AUI_CTU**: Instanz des Typs adapter::events::unidirectional::AUI_CTU.
- **DigitalInput_CLK_I2**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I2
  - Parameter InputEvent = BUTTON_SINGLE_CLICK

Verbindungen und Schnittstellen

Adapterverbindungen:

- AUI_CTU.Q -> DigitalOutput_Q1.OUT

Ereignisverbindungen:

- DigitalInput_CLK_I1.IND -> AUI_CTU.CU
- DigitalInput_CLK_I2.IND -> AUI_CTU.R

Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.

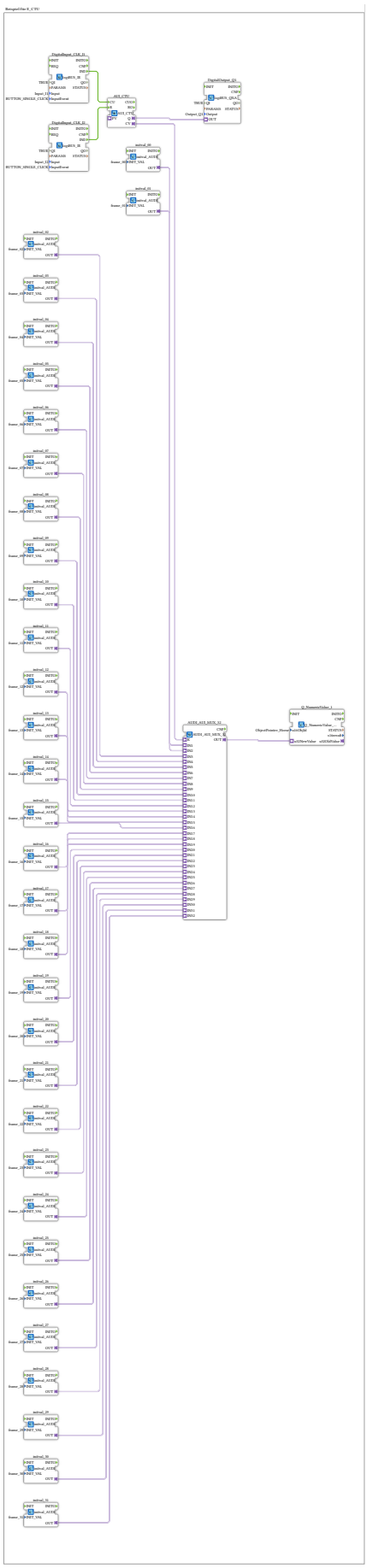
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_080e_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.381 Uebung_080f_AX: Beispiel für E_CTU

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_080f_AX (Beispiel für E_CTU).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Beispiel für E_CTU**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_080f_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
- **DigitalInput_CLK_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **AUI_CTU**: Instanz des Typs adapter::events::unidirectional::AUI_CTU.
- **DigitalInput_CLK_I2**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I2
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **initval_00**: Instanz des Typs adapter::types::unidirectional::AUDI::initval::initval_AUDI.
  - Parameter INIT_VAL = frame_00
- **initval_01**: Instanz des Typs adapter::types::unidirectional::AUDI::initval::initval_AUDI.
  - Parameter INIT_VAL = frame_01
- **initval_02**: Instanz des Typs adapter::types::unidirectional::AUDI::initval::initval_AUDI.
  - Parameter INIT_VAL = frame_02
- **initval_03**: Instanz des Typs adapter::types::unidirectional::AUDI::initval::initval_AUDI.
  - Parameter INIT_VAL = frame_03
- **initval_04**: Instanz des Typs adapter::types::unidirectional::AUDI::initval::initval_AUDI.
  - Parameter INIT_VAL = frame_04
- **initval_05**: Instanz des Typs adapter::types::unidirectional::AUDI::initval::initval_AUDI.
  - Parameter INIT_VAL = frame_05
- **initval_06**: Instanz des Typs adapter::types::unidirectional::AUDI::initval::initval_AUDI.
  - Parameter INIT_VAL = frame_06
- **initval_07**: Instanz des Typs adapter::types::unidirectional::AUDI::initval::initval_AUDI.
  - Parameter INIT_VAL = frame_07
- **initval_08**: Instanz des Typs adapter::types::unidirectional::AUDI::initval::initval_AUDI.
  - Parameter INIT_VAL = frame_08
- **initval_09**: Instanz des Typs adapter::types::unidirectional::AUDI::initval::initval_AUDI.
  - Parameter INIT_VAL = frame_09
- **initval_10**: Instanz des Typs adapter::types::unidirectional::AUDI::initval::initval_AUDI.
  - Parameter INIT_VAL = frame_10
- **initval_11**: Instanz des Typs adapter::types::unidirectional::AUDI::initval::initval_AUDI.
  - Parameter INIT_VAL = frame_11
- **initval_12**: Instanz des Typs adapter::types::unidirectional::AUDI::initval::initval_AUDI.
  - Parameter INIT_VAL = frame_12
- **initval_13**: Instanz des Typs adapter::types::unidirectional::AUDI::initval::initval_AUDI.
  - Parameter INIT_VAL = frame_13
- **initval_14**: Instanz des Typs adapter::types::unidirectional::AUDI::initval::initval_AUDI.
  - Parameter INIT_VAL = frame_14
- **initval_15**: Instanz des Typs adapter::types::unidirectional::AUDI::initval::initval_AUDI.
  - Parameter INIT_VAL = frame_15
- **initval_16**: Instanz des Typs adapter::types::unidirectional::AUDI::initval::initval_AUDI.
  - Parameter INIT_VAL = frame_16
- **initval_17**: Instanz des Typs adapter::types::unidirectional::AUDI::initval::initval_AUDI.
  - Parameter INIT_VAL = frame_17
- **initval_18**: Instanz des Typs adapter::types::unidirectional::AUDI::initval::initval_AUDI.
  - Parameter INIT_VAL = frame_18
- **initval_19**: Instanz des Typs adapter::types::unidirectional::AUDI::initval::initval_AUDI.

  - Parameter INIT_VAL = frame_19
- **initval_20**: Instanz des Typs adapter::types::unidirectional::AUDI::initval::initval_AUDI.
  - Parameter INIT_VAL = frame_20
- **initval_21**: Instanz des Typs adapter::types::unidirectional::AUDI::initval::initval_AUDI.
  - Parameter INIT_VAL = frame_21
- **initval_22**: Instanz des Typs adapter::types::unidirectional::AUDI::initval::initval_AUDI.
  - Parameter INIT_VAL = frame_22
- **initval_23**: Instanz des Typs adapter::types::unidirectional::AUDI::initval::initval_AUDI.
  - Parameter INIT_VAL = frame_23
- **initval_24**: Instanz des Typs adapter::types::unidirectional::AUDI::initval::initval_AUDI.
  - Parameter INIT_VAL = frame_24
- **initval_25**: Instanz des Typs adapter::types::unidirectional::AUDI::initval::initval_AUDI.
  - Parameter INIT_VAL = frame_25
- **initval_26**: Instanz des Typs adapter::types::unidirectional::AUDI::initval::initval_AUDI.
  - Parameter INIT_VAL = frame_26
- **initval_27**: Instanz des Typs adapter::types::unidirectional::AUDI::initval::initval_AUDI.
  - Parameter INIT_VAL = frame_27
- **initval_28**: Instanz des Typs adapter::types::unidirectional::AUDI::initval::initval_AUDI.
  - Parameter INIT_VAL = frame_28
- **initval_29**: Instanz des Typs adapter::types::unidirectional::AUDI::initval::initval_AUDI.
  - Parameter INIT_VAL = frame_29
- **initval_30**: Instanz des Typs adapter::types::unidirectional::AUDI::initval::initval_AUDI.
  - Parameter INIT_VAL = frame_30
- **initval_31**: Instanz des Typs adapter::types::unidirectional::AUDI::initval::initval_AUDI.
  - Parameter INIT_VAL = frame_31
- **AUDI_AUI_MUX_32**: Instanz des Typs adapter::selection::unidirectional::AUDI_AUI_MUX_32.
- **Q_NumericValue_1**: Instanz des Typs isobus::UT::Q::Q_NumericValue_AUDI.
  - Parameter u16ObjId = ObjectPointer_Horse

Verbindungen und Schnittstellen

Adapterverbindungen:

- AUI_CTU.Q -> DigitalOutput_Q1.OUT
- AUI_CTU.CV -> AUDI_AUI_MUX_32.K
- initval_00.OUT -> AUDI_AUI_MUX_32.IN1
- initval_01.OUT -> AUDI_AUI_MUX_32.IN2
- initval_02.OUT -> AUDI_AUI_MUX_32.IN3
- initval_03.OUT -> AUDI_AUI_MUX_32.IN4
- initval_04.OUT -> AUDI_AUI_MUX_32.IN5
- initval_05.OUT -> AUDI_AUI_MUX_32.IN6
- initval_06.OUT -> AUDI_AUI_MUX_32.IN7
- initval_07.OUT -> AUDI_AUI_MUX_32.IN8
- initval_08.OUT -> AUDI_AUI_MUX_32.IN9
- initval_09.OUT -> AUDI_AUI_MUX_32.IN10
- initval_10.OUT -> AUDI_AUI_MUX_32.IN11
- initval_11.OUT -> AUDI_AUI_MUX_32.IN12
- initval_12.OUT -> AUDI_AUI_MUX_32.IN13
- initval_13.OUT -> AUDI_AUI_MUX_32.IN14
- initval_14.OUT -> AUDI_AUI_MUX_32.IN15
- initval_15.OUT -> AUDI_AUI_MUX_32.IN16
- initval_16.OUT -> AUDI_AUI_MUX_32.IN17
- initval_17.OUT -> AUDI_AUI_MUX_32.IN18
- initval_18.OUT -> AUDI_AUI_MUX_32.IN19
- initval_19.OUT -> AUDI_AUI_MUX_32.IN20
- initval_20.OUT -> AUDI_AUI_MUX_32.IN21
- initval_21.OUT -> AUDI_AUI_MUX_32.IN22
- initval_22.OUT -> AUDI_AUI_MUX_32.IN23
- initval_23.OUT -> AUDI_AUI_MUX_32.IN24
- initval_24.OUT -> AUDI_AUI_MUX_32.IN25
- initval_25.OUT -> AUDI_AUI_MUX_32.IN26

- initval_26.OUT -> AUDI_AUI_MUX_32.IN27
- initval_27.OUT -> AUDI_AUI_MUX_32.IN28
- initval_28.OUT -> AUDI_AUI_MUX_32.IN29
- initval_29.OUT -> AUDI_AUI_MUX_32.IN30
- initval_30.OUT -> AUDI_AUI_MUX_32.IN31
- initval_31.OUT -> AUDI_AUI_MUX_32.IN32
- AUDI_AUI_MUX_32.OUT -> Q_NumericValue_1.u32NewValue

Ereignisverbindungen:

- DigitalInput_CLK_I1.IND -> AUI_CTU.CU
- DigitalInput_CLK_I2.IND -> AUI_CTU.R

Programmablauf und Funktionsweise

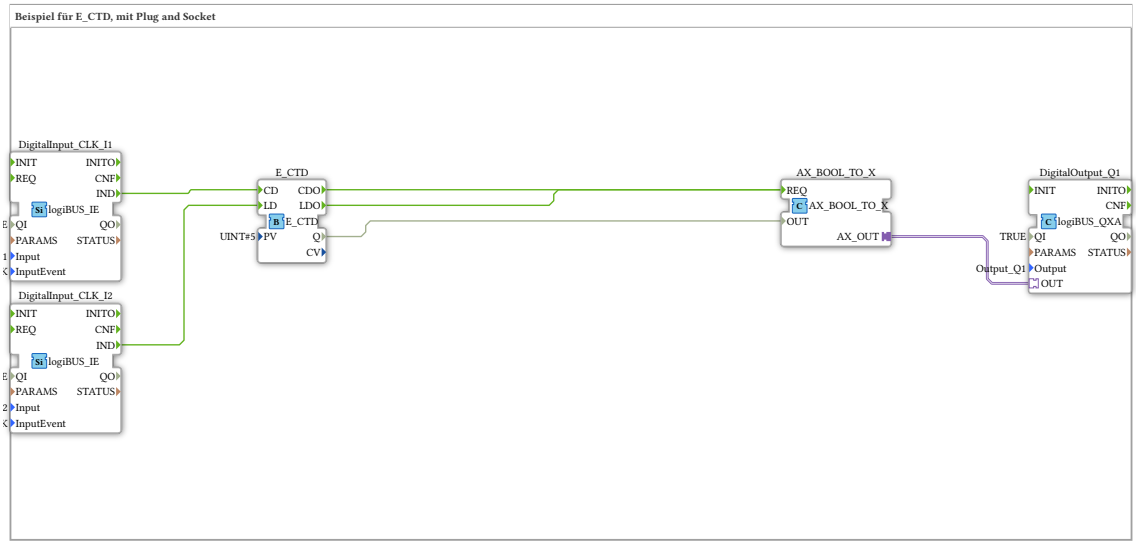
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_080f_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.382 Uebung_081_AX: Beispiel für E_CTD, mit Plug and Socket

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_081_AX (Beispiel für E_CTD, mit Plug and Socket).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Beispiel für E_CTD, mit Plug and Socket**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_081_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1

- **DigitalInput_CLK_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **E_CTD**: Instanz des Typs iec61499::events::E_CTD.
  - Parameter PV = UINT#5
- **DigitalInput_CLK_I2**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I2
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **AX_BOOL_TO_X**: Instanz des Typs adapter::conversion::unidirectional::AX_BOOL_TO_X.

Verbindungen und Schnittstellen

Adapterverbindungen:

- AX_BOOL_TO_X.AX_OUT -> DigitalOutput_Q1.OUT

Ereignisverbindungen:

- DigitalInput_CLK_I1.IND -> E_CTD.CD
- E_CTD.CDO -> AX_BOOL_TO_X.REQ
- E_CTD.LDO -> AX_BOOL_TO_X.REQ
- DigitalInput_CLK_I2.IND -> E_CTD.LD

Datenverbindungen:

- E_CTD.Q -> AX_BOOL_TO_X.OUT

Programmablauf und Funktionsweise

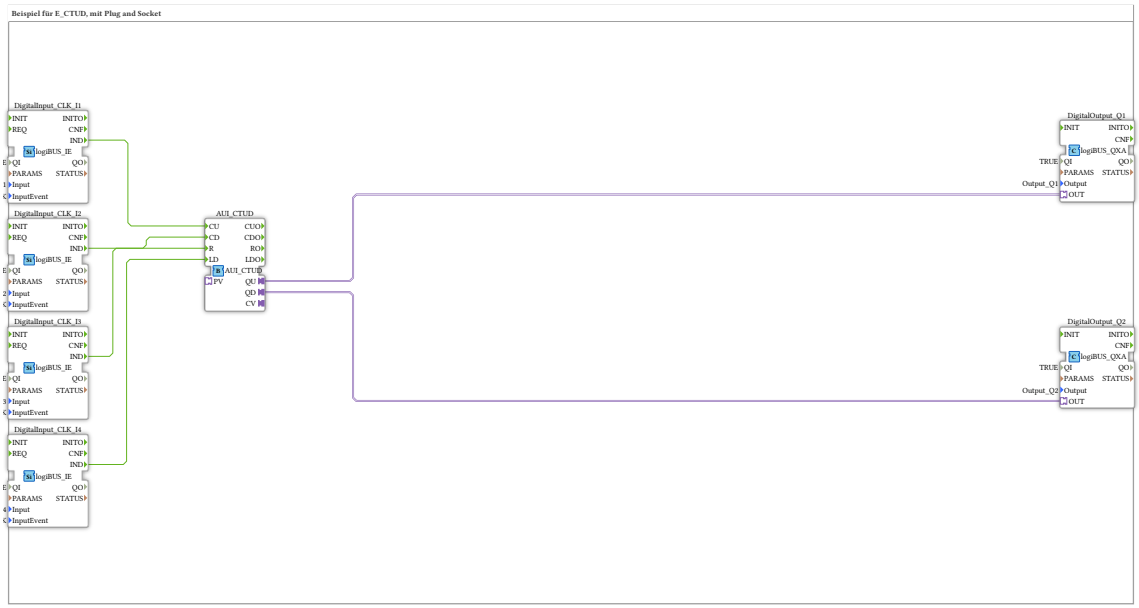
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_081_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.383 Uebung_082_AX: Beispiel für E_CTUD, mit Plug and Socket

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_082_AX (Beispiel für E_CTUD, mit Plug and Socket).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Beispiel für E_CTUD, mit Plug and Socket**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_082_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
- **DigitalInput_CLK_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **AUI_CTUD**: Instanz des Typs adapter::events::unidirectional::AUI_CTUD.
- **DigitalInput_CLK_I2**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I2
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **DigitalInput_CLK_I3**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I3
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **DigitalInput_CLK_I4**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I4
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **DigitalOutput_Q2**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q2

Verbindungen und Schnittstellen

Adapterverbindungen:

- AUI_CTUD.QU -> DigitalOutput_Q1.OUT
- AUI_CTUD.QD -> DigitalOutput_Q2.OUT

Ereignisverbindungen:

- DigitalInput_CLK_I1.IND -> AUI_CTUD.CU
- DigitalInput_CLK_I2.IND -> AUI_CTUD.CD
- DigitalInput_CLK_I3.IND -> AUI_CTUD.R
- DigitalInput_CLK_I4.IND -> AUI_CTUD.LD

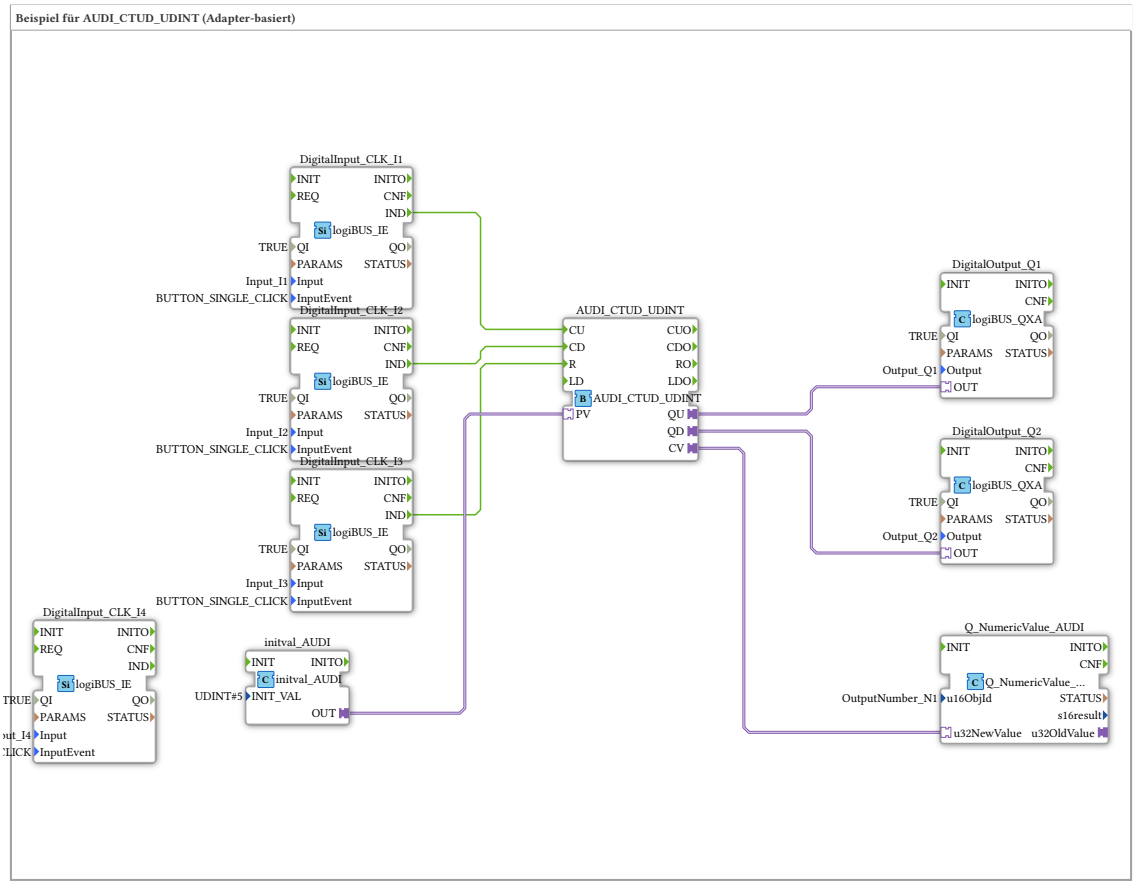
Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_082_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.384 Uebung_083_AUDI: Beispiel für AUDI_CTUD_UDINT (Adapter-basiert)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert den Aufwärts-/Abwärtszähler `AUDI_CTUD_UDINT` auf Basis von Adaptern. Drei Taster mit Single-Click-Erkennung steuern den Zähler: `I1` zählt hoch (`CU`), `I2` zählt runter (`CD`), `I3` setzt zurück (`R`). Der Vorgabewert `PV` kommt fest aus `initval_AUDI` (`INIT_VAL = 5`). Der Zählerwert `CV` wird an der Anzeige `OutputNumber_N1` ausgegeben, die Ausgänge `QU` und `QD` schalten `Q1` und `Q2`. Der Baustein `DigitalInput_CLK_I4` (Eingang `I4`) ist im Netzwerk vorhanden, aber nicht verdrahtet.

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_CLK_I1**, **DigitalInput_CLK_I2**, **DigitalInput_CLK_I3**, **DigitalInput_CLK_I4**: Taster mit Klick-Erkennung
  - **Typ**: `logiBUS::io::DI::logiBUS_IE`
  - **Parameter**: `QI = TRUE`, `Input = Input_I1` / `Input_I2` / `Input_I3` / `Input_I4`, jeweils `InputEvent = BUTTON_SINGLE_CLICK`
  - **Erklärung**: Melden einen Einzelklick auf den jeweiligen Eingang als Event `IND`.
- **AUDI_CTUD_UDINT**: Adapter-basierter Auf-/Abwärtszähler für `UDINT`
  - **Typ**: `adapter::events::unidirectional::AUDI_CTUD_UDINT`
  - **Parameter**: keine
  - **Erklärung**: Ereigniseingänge `CU` (hoch), `CD` (runter), `R` (Reset). `CV` ist der aktuelle Zählwert (Adapter), `QU` ist `TRUE`, wenn `CV` größer oder gleich `PV` ist, `QD` ist `TRUE`, wenn `CV` kleiner oder gleich 0 ist.
- **initval_AUDI**: Vorgabewert
  - **Typ**: `adapter::types::unidirectional::AUDI::initval::initval_AUDI`
  - **Parameter**: `INIT_VAL = 5`
  - **Erklärung**: Liefert den Wert 5 als Vorgabewert `PV` an den Zähler.
- **DigitalOutput_Q1**, **DigitalOutput_Q2**: logiBUS Digitalausgänge
  - **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
  - **Parameter**: `QI = TRUE`, `Output = Output_Q1` bzw. `Output_Q2`
  - **Erklärung**: `Q1` zeigt `QU`, `Q2` zeigt `QD`.
- **Q_NumericValue_AUDI**: numerische Anzeige
  - **Typ**: `isobus::UT::Q::Q_NumericValue_AUDI`
  - **Parameter**: `u16ObjId = OutputNumber_N1`
  - **Erklärung**: Zeigt den Zählerwert `CV` (`u32NewValue`) an `OutputNumber_N1` an.

Programmablauf und Verbindungen

1. `DigitalInput_CLK_I1.IND` → `AUDI_CTUD_UDINT.CU`, `DigitalInput_CLK_I2.IND` → `AUDI_CTUD_UDINT.CD` und `DigitalInput_CLK_I3.IND` → `AUDI_CTUD_UDINT.R` (EventConnections): Ein Klick auf `I1` zählt hoch, auf `I2` runter, auf `I3` wird der Zähler zurückgesetzt.
2. `initval_AUDI.OUT` → `AUDI_CTUD_UDINT.PV` (AdapterConnection): Der Vorgabewert 5.
3. `AUDI_CTUD_UDINT.QU` → `DigitalOutput_Q1.OUT` und `AUDI_CTUD_UDINT.QD` → `DigitalOutput_Q2.OUT` (AdapterConnections): `Q1` ist an, wenn der Zählwert den Vorgabewert erreicht oder überschreitet; `Q2` ist an, wenn der Zählwert 0 (oder kleiner) ist.
4. `AUDI_CTUD_UDINT.CV` → `Q_NumericValue_AUDI.u32NewValue` (AdapterConnection): Der Zählerwert wird numerisch angezeigt.

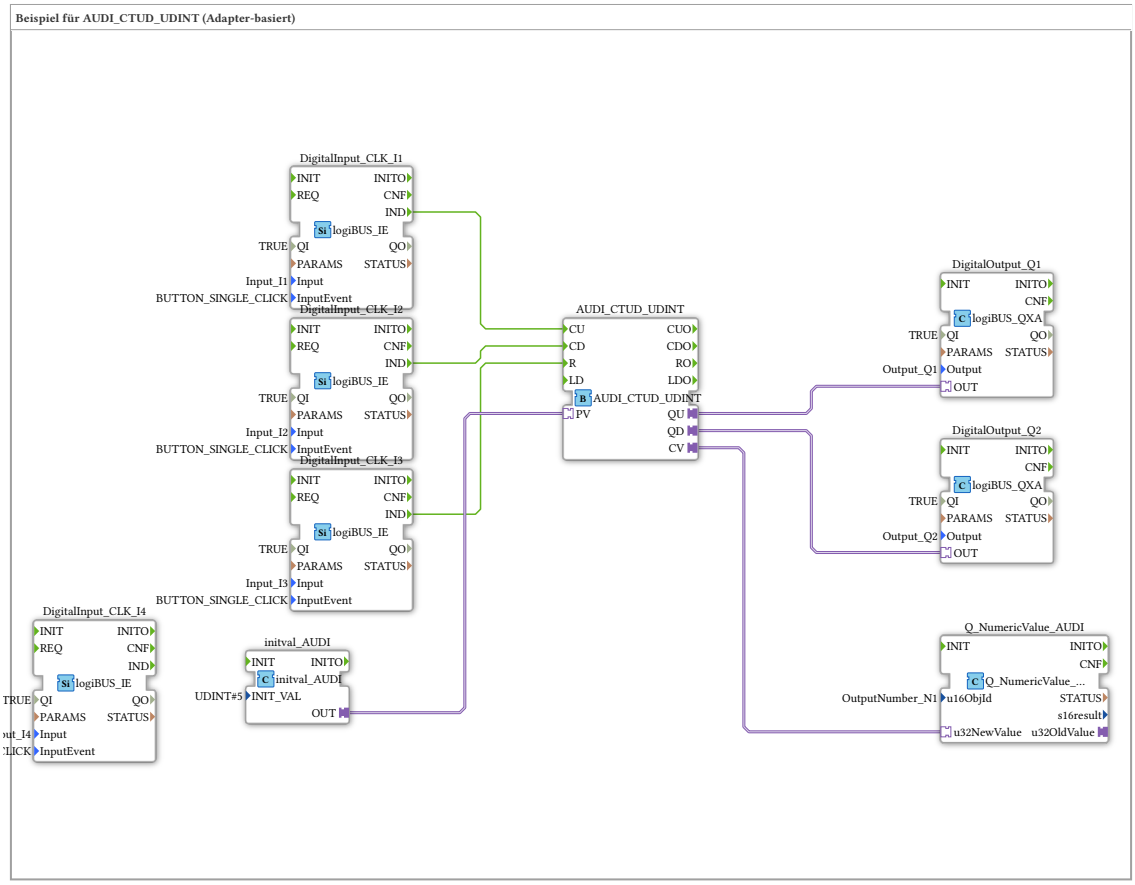
Zusammenfassung

Die Übung veranschaulicht den Einsatz des Adapter-basierten Auf-/Abwärtszählers `AUDI_CTUD_UDINT`. Drei Taster zählen hoch, zählen runter und setzen zurück; der Vorgabewert 5 kommt aus `initval_AUDI`. `QU` und `QD` werden auf `Q1` und `Q2` geführt, der aktuelle Zählerwert wird an `OutputNumber_N1` angezeigt.

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

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

2.1.2.385 Uebung_083_AX: Beispiel für AUDI_CTUD_UDINT (Adapter-basiert)



• – ○ • – ○ • – ○ •

Einleitung

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_083_AX`. Sie ist die adapterbasierte Variante der Übung 083 und demonstriert einen Auf-/Abwärtszähler (`AUDI_CTUD_UDINT`) für vorzeichenlose 32-Bit-Ganzzahlen. Ziel ist, die adapterbasierte Steuerung eines Zählers und die Anzeige des Zählwerts auf einem ISOBUS-Terminal kennenzulernen.

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_CLK_I1**: Taster „Zähler hoch“
  - **Typ**: `logiBUS::io::DI::logiBUS_IE`
  - **Parameter**: `QI = TRUE`, `Input = Input_I1`, `InputEvent = BUTTON_SINGLE_CLICK`
  - **Erklärung**: Einzelklick auf `I1` ⟶ `AUDI_CTUD_UDINT.CU`.
- **DigitalInput_CLK_I2**: Taster „Zähler runter“
  - **Typ**: `logiBUS::io::DI::logiBUS_IE`
  - **Parameter**: `QI = TRUE`, `Input = Input_I2`, `InputEvent = BUTTON_SINGLE_CLICK`
  - **Erklärung**: Einzelklick auf `I2` ⟶ `AUDI_CTUD_UDINT.CD`.
- **DigitalInput_CLK_I3**: Taster „Reset“
  - **Typ**: `logiBUS::io::DI::logiBUS_IE`
  - **Parameter**: `QI = TRUE`, `Input = Input_I3`, `InputEvent = BUTTON_SINGLE_CLICK`
  - **Erklärung**: Einzelklick auf `I3` ⟶ `AUDI_CTUD_UDINT.R`.
- **DigitalInput_CLK_I4**: Eingang `I4`
  - **Typ**: `logiBUS::io::DI::logiBUS_IE`
  - **Parameter**: `QI = TRUE`, `Input = Input_I4`, `InputEvent = BUTTON_SINGLE_CLICK`
  - **Erklärung**: Ist im Netzwerk instanziiert, aber mit keinem anderen Baustein verbunden (insbesondere nicht mit `LD`).
- **initval_AUDI**: Startwert für `PV`
  - **Typ**: `adapter::types::unidirectional::AUDI::initval::initval_AUDI`

  - **Parameter**: `INIT_VAL = 5`
  - **Erklärung**: Liefert beim Initialisieren den UDINT-Wert `5` als Adapter `OUT`, der an `AUDI_CTUD_UDINT.PV` geht.
- **AUDI_CTUD_UDINT**: Adapterbasierter Auf-/Abwärtszähler
  - **Typ**: `adapter::events::unidirectional::AUDI_CTUD_UDINT`
  - **Parameter**: keine
  - **Erklärung**: Ereignisse `CU` (hoch), `CD` (runter), `R` (Reset auf 0) und `LD` (CV mit PV laden; hier nicht verdrahtet). Adapterausgänge: `QU` (TRUE, wenn `CV >= PV`), `QD` (TRUE, wenn `CV <= 0`) und `CV` (Zählwert). `PV` ist ein Adaptereingang. Die Ausgänge werden nur bei Änderung gemeldet.
- **DigitalOutput_Q1**: Digitalausgang für `QU`
  - **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
  - **Parameter**: `QI = TRUE`, `Output = Output_Q1`
  - **Erklärung**: Schaltet `Q1` gemäß `AUDI_CTUD_UDINT.QU`.
- **DigitalOutput_Q2**: Digitalausgang für `QD`
  - **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
  - **Parameter**: `QI = TRUE`, `Output = Output_Q2`
  - **Erklärung**: Schaltet `Q2` gemäß `AUDI_CTUD_UDINT.QD`.
- **Q_NumericValue_AUDI**: Anzeige am Terminal
  - **Typ**: `isobus::UT::Q::Q_NumericValue_AUDI`
  - **Parameter**: `u16ObjId = OutputNumber_N1`
  - **Erklärung**: Zeigt den Zählwert über den `AUDI`-Adapter (`u32NewValue`) im VT-Zahlenfeld `OutputNumber_N1` an.

Programmablauf und Verbindungen

1. `DigitalInput_CLK_I1.IND` ⟶ `AUDI_CTUD_UDINT.CU`: jeder Klick erhöht den Zähler um 1.
2. `DigitalInput_CLK_I2.IND` ⟶ `AUDI_CTUD_UDINT.CD`: jeder Klick verringert den Zähler um 1.
3. `DigitalInput_CLK_I3.IND` ⟶ `AUDI_CTUD_UDINT.R`: setzt den Zähler auf 0 zurück.
4. `initval_AUDI.OUT` ⟶ `AUDI_CTUD_UDINT.PV`: der Vorgabewert ist 5.
5. `AUDI_CTUD_UDINT.QU` ⟶ `DigitalOutput_Q1.OUT` und `AUDI_CTUD_UDINT.QD` ⟶ `DigitalOutput_Q2.OUT`: Statusanzeige des Zählers.
6. `AUDI_CTUD_UDINT.CV` ⟶ `Q_NumericValue_AUDI.u32NewValue`: der Zählwert wird am Terminal angezeigt.

Verhalten: `Q1` ist EIN, sobald der Zählwert den Vorgabewert 5 erreicht oder überschreitet; `Q2` ist EIN, solange der Zählwert 0 ist.

Zusammenfassung

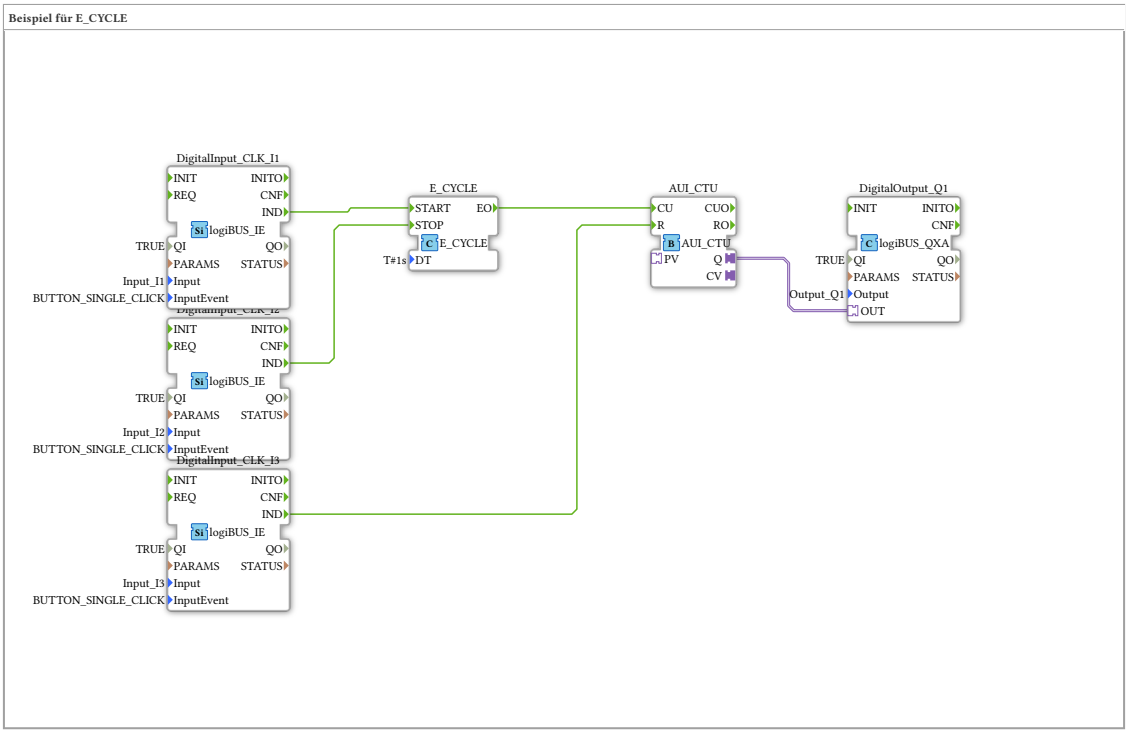
Die Übung zeigt einen vollständig adapterbasierten Zähler: Ereignisse von Tastern steuern `AUDI_CTUD_UDINT`, der Vorgabewert wird über einen `AUDI`-Adapter (`initval_AUDI`) geliefert, und Statusausgänge sowie Zählwert werden ohne zusätzliche Datenverdrahtung an Ausgänge und das Terminal übergeben.

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

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

2.1.2.386 Uebung_084_AX: Beispiel für E_CYCLE

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_084_AX (Beispiel für E_CYCLE).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Beispiel für E_CYCLE**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_084_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
- **AUI_CTU**: Instanz des Typs adapter::events::unidirectional::AUI_CTU.
- **E_CYCLE**: Instanz des Typs iec61499::events::E_CYCLE.
  - Parameter DT = T#1s
- **DigitalInput_CLK_I2**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I2
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **DigitalInput_CLK_I3**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I3
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **DigitalInput_CLK_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
  - Parameter InputEvent = BUTTON_SINGLE_CLICK

Verbindungen und Schnittstellen

Adapterverbindungen:

- AUI_CTU.Q -> DigitalOutput_Q1.OUT

Ereignisverbindungen:

- E_CYCLE.EO -> AUI_CTU.CU

- DigitalInput_CLK_I1.IND -> E_CYCLE.START
- DigitalInput_CLK_I2.IND -> E_CYCLE.STOP
- DigitalInput_CLK_I3.IND -> AUI_CTU.R

Programmablauf und Funktionsweise

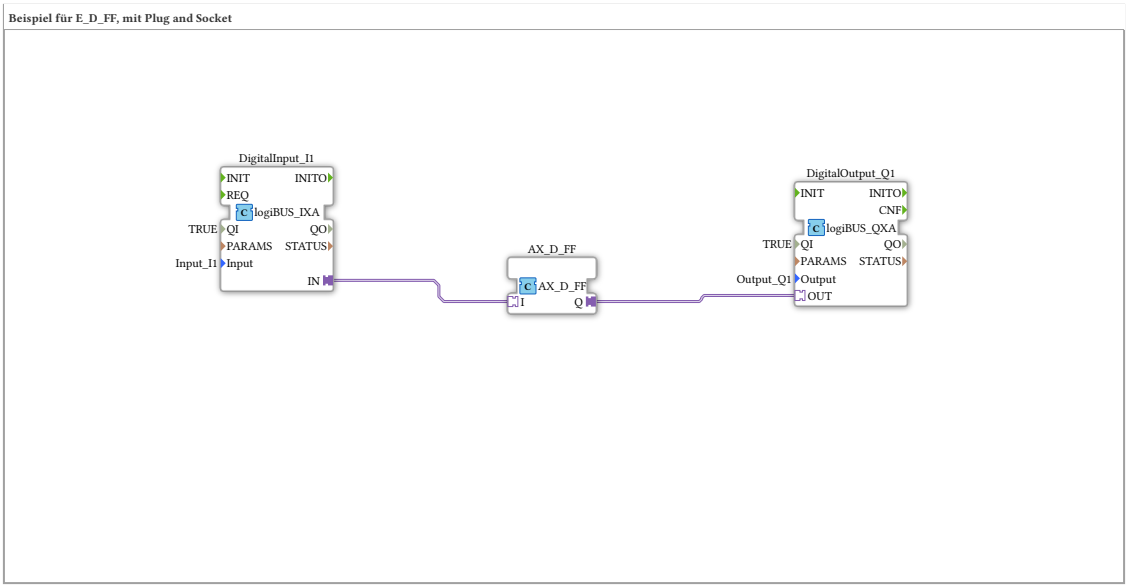
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_084_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.387 Uebung_085_AX: Beispiel für E_D_FF, mit Plug and Socket

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_085_AX (Beispiel für E_D_FF, mit Plug and Socket).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Beispiel für E_D_FF, mit Plug and Socket**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_085_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
- **DigitalInput_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
- **AX_D_FF**: Instanz des Typs adapter::events::unidirectional::AX_D_FF.

Verbindungen und Schnittstellen

Adapterverbindungen:

- DigitalInput_I1.IN -> AX_D_FF.I
- AX_D_FF.Q -> DigitalOutput_Q1.OUT

Programmablauf und Funktionsweise

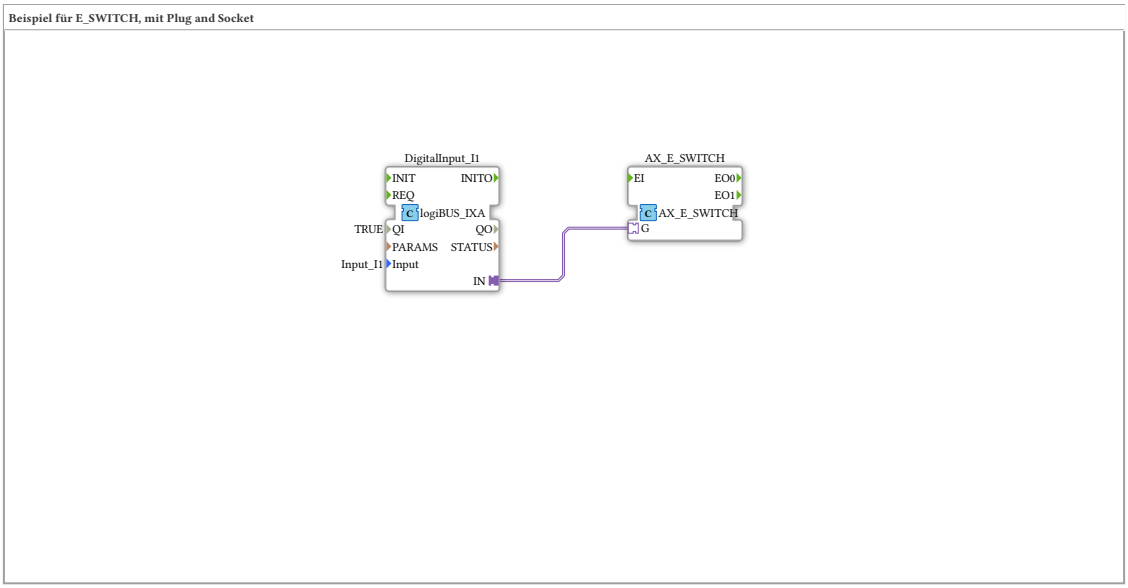
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_085_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.388 Uebung_086_AX: Beispiel für E_SWITCH, mit Plug and Socket

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_086_AX (Beispiel für E_SWITCH, mit Plug and Socket).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Beispiel für E_SWITCH, mit Plug and Socket**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_086_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
- **AX_E_SWITCH**: Instanz des Typs adapter::events::unidirectional::AX_E_SWITCH.

Verbindungen und Schnittstellen

Adapterverbindungen:

- DigitalInput_I1.IN -> AX_E_SWITCH.G

Programmablauf und Funktionsweise

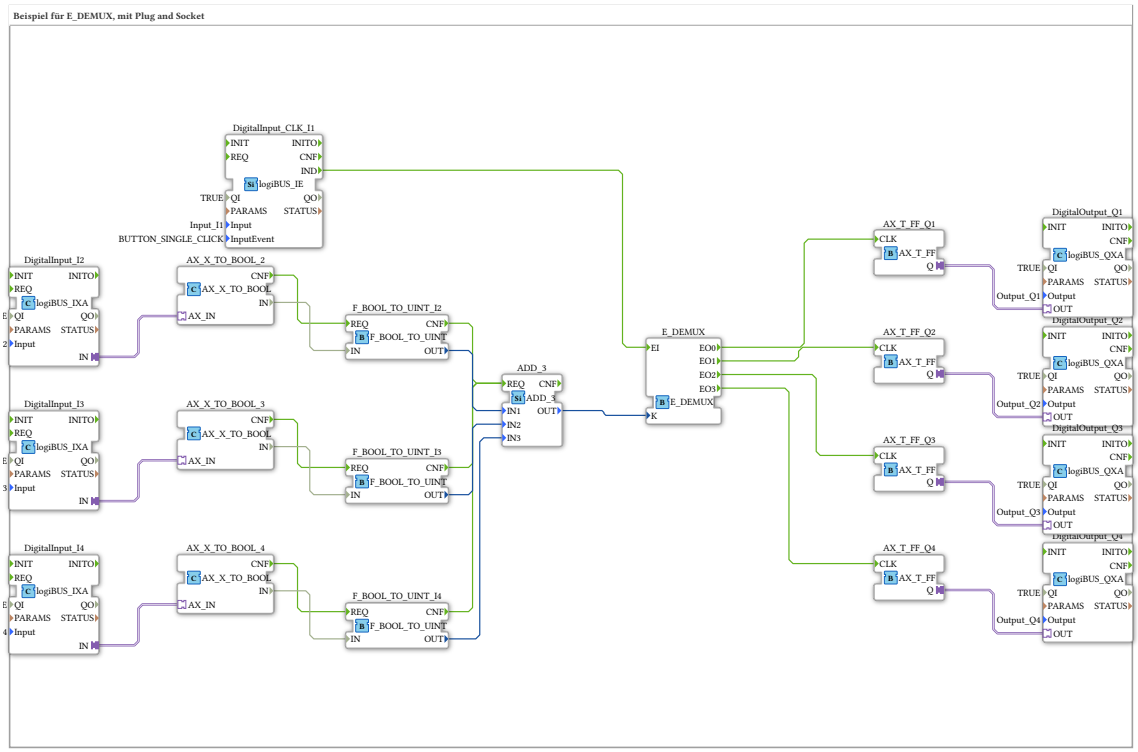
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_086_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.389 Uebung_087_AX: Beispiel für E_DEMUX, mit Plug and Socket

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_087_AX (Beispiel für E_DEMUX, mit Plug and Socket).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Beispiel für E_DEMUX, mit Plug and Socket**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_087_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **E_DEMUX**: Instanz des Typs iec61499::events::E_DEMUX.
- **ADD_3**: Instanz des Typs iec61131::arithmetic::ADD_3.
- **DigitalInput_I2**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I2
- **DigitalInput_I3**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.

  - Parameter QI = TRUE
  - Parameter Input = Input_I3
- **DigitalInput_I4**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I4
- **AX_X_TO_BOOL_2**: Instanz des Typs adapter::conversion::unidirectional::AX_X_TO_BOOL.
- **AX_X_TO_BOOL_3**: Instanz des Typs adapter::conversion::unidirectional::AX_X_TO_BOOL.
- **AX_X_TO_BOOL_4**: Instanz des Typs adapter::conversion::unidirectional::AX_X_TO_BOOL.
- **F_BOOL_TO_UINT_I2**: Instanz des Typs iec61131::conversion::F_BOOL_TO_UINT.
- **F_BOOL_TO_UINT_I3**: Instanz des Typs iec61131::conversion::F_BOOL_TO_UINT.
- **F_BOOL_TO_UINT_I4**: Instanz des Typs iec61131::conversion::F_BOOL_TO_UINT.
- **DigitalOutput_Q2**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q2
- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
- **DigitalOutput_Q4**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q4
- **DigitalOutput_Q3**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q3
- **AX_T_FF_Q1**: Instanz des Typs adapter::events::unidirectional::AX_T_FF.
- **AX_T_FF_Q2**: Instanz des Typs adapter::events::unidirectional::AX_T_FF.
- **AX_T_FF_Q3**: Instanz des Typs adapter::events::unidirectional::AX_T_FF.
- **AX_T_FF_Q4**: Instanz des Typs adapter::events::unidirectional::AX_T_FF.
- **DigitalInput_CLK_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
  - Parameter InputEvent = BUTTON_SINGLE_CLICK

Verbindungen und Schnittstellen

Adapterverbindungen:

- DigitalInput_I2.IN -> AX_X_TO_BOOL_2.AX_IN
- DigitalInput_I3.IN -> AX_X_TO_BOOL_3.AX_IN
- DigitalInput_I4.IN -> AX_X_TO_BOOL_4.AX_IN
- AX_T_FF_Q1.Q -> DigitalOutput_Q1.OUT
- AX_T_FF_Q2.Q -> DigitalOutput_Q2.OUT
- AX_T_FF_Q3.Q -> DigitalOutput_Q3.OUT
- AX_T_FF_Q4.Q -> DigitalOutput_Q4.OUT

Ereignisverbindungen:

- AX_X_TO_BOOL_2.CNF -> F_BOOL_TO_UINT_I2.REQ
- AX_X_TO_BOOL_3.CNF -> F_BOOL_TO_UINT_I3.REQ
- AX_X_TO_BOOL_4.CNF -> F_BOOL_TO_UINT_I4.REQ
- F_BOOL_TO_UINT_I4.CNF -> ADD_3.REQ
- F_BOOL_TO_UINT_I3.CNF -> ADD_3.REQ
- F_BOOL_TO_UINT_I2.CNF -> ADD_3.REQ
- E_DEMUX.EO0 -> AX_T_FF_Q1.CLK
- E_DEMUX.EO1 -> AX_T_FF_Q2.CLK
- E_DEMUX.EO2 -> AX_T_FF_Q3.CLK
- E_DEMUX.EO3 -> AX_T_FF_Q4.CLK
- DigitalInput_CLK_I1.IND -> E_DEMUX.EI

Datenverbindungen:

- AX_X_TO_BOOL_2.IN -> F_BOOL_TO_UINT_I2.IN
- AX_X_TO_BOOL_3.IN -> F_BOOL_TO_UINT_I3.IN
- AX_X_TO_BOOL_4.IN -> F_BOOL_TO_UINT_I4.IN

- F_BOOL_TO_UINT_I2.OUT -> ADD_3.IN1
- F_BOOL_TO_UINT_I3.OUT -> ADD_3.IN2
- F_BOOL_TO_UINT_I4.OUT -> ADD_3.IN3
- ADD_3.OUT -> E_DEMUX.K

Hinweise aus dem Modell

Diese Taste schaltet ein oder aus Die Anzahl der restlichen Tasten bestimmt den Ausgang: Keine –> Q1 Eine –> Q2 Zwei –> Q3 Drei –> Q4

Programmablauf und Funktionsweise

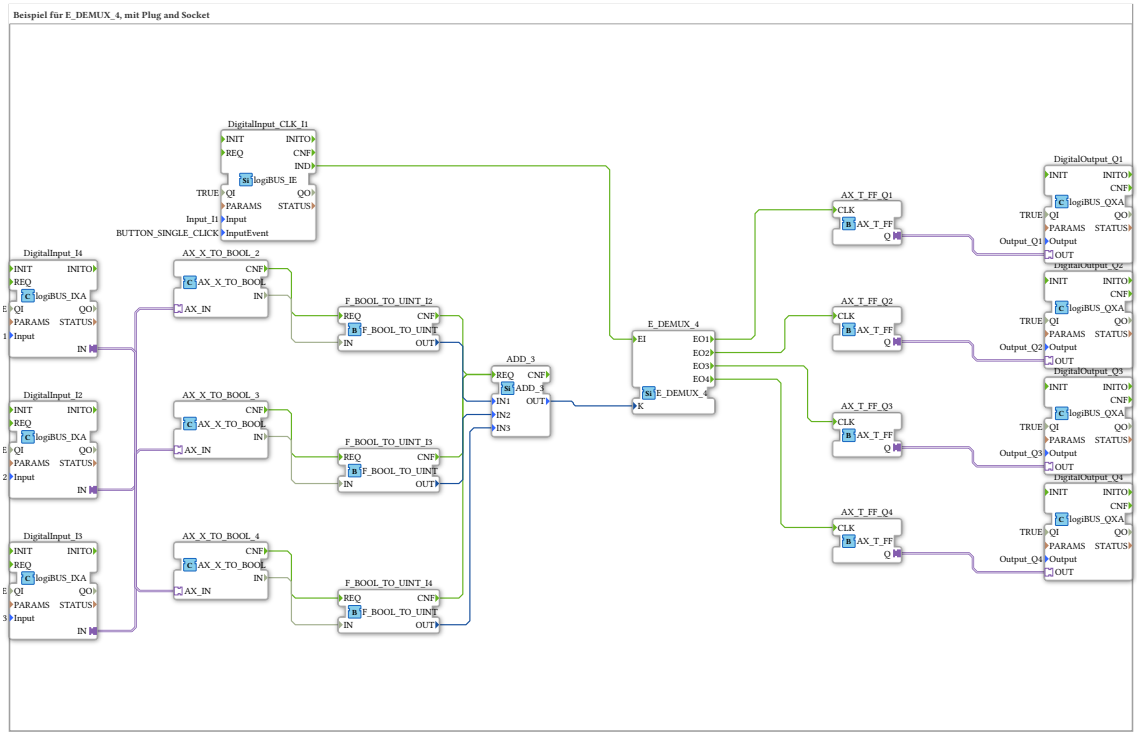
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_087_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.390 Uebung_087a1_AX: Beispiel für E_DEMUX_4, mit Plug and Socket

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_087a1_AX (Beispiel für E_DEMUX_4, mit Plug and Socket).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Beispiel für E_DEMUX_4, mit Plug and Socket**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_087a1_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **E_DEMUX_4**: Instanz des Typs iec61499::events::E_DEMUX_4.
- **ADD_3**: Instanz des Typs iec61131::arithmetic::ADD_3.
- **DigitalInput_I2**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I2
- **DigitalInput_I3**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I3
- **DigitalInput_I4**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
- **AX_X_TO_BOOL_2**: Instanz des Typs adapter::conversion::unidirectional::AX_X_TO_BOOL.
- **AX_X_TO_BOOL_3**: Instanz des Typs adapter::conversion::unidirectional::AX_X_TO_BOOL.
- **AX_X_TO_BOOL_4**: Instanz des Typs adapter::conversion::unidirectional::AX_X_TO_BOOL.
- **F_BOOL_TO_UINT_I2**: Instanz des Typs iec61131::conversion::F_BOOL_TO_UINT.
- **F_BOOL_TO_UINT_I3**: Instanz des Typs iec61131::conversion::F_BOOL_TO_UINT.
- **F_BOOL_TO_UINT_I4**: Instanz des Typs iec61131::conversion::F_BOOL_TO_UINT.
- **DigitalOutput_Q2**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q2
- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
- **DigitalOutput_Q4**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q4
- **DigitalOutput_Q3**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q3
- **AX_T_FF_Q1**: Instanz des Typs adapter::events::unidirectional::AX_T_FF.
- **AX_T_FF_Q2**: Instanz des Typs adapter::events::unidirectional::AX_T_FF.
- **AX_T_FF_Q3**: Instanz des Typs adapter::events::unidirectional::AX_T_FF.
- **AX_T_FF_Q4**: Instanz des Typs adapter::events::unidirectional::AX_T_FF.
- **DigitalInput_CLK_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
  - Parameter InputEvent = BUTTON_SINGLE_CLICK

Verbindungen und Schnittstellen

Adapterverbindungen:

- DigitalInput_I2.IN -> AX_X_TO_BOOL_2.AX_IN
- DigitalInput_I3.IN -> AX_X_TO_BOOL_3.AX_IN
- DigitalInput_I4.IN -> AX_X_TO_BOOL_4.AX_IN
- AX_T_FF_Q1.Q -> DigitalOutput_Q1.OUT
- AX_T_FF_Q2.Q -> DigitalOutput_Q2.OUT
- AX_T_FF_Q3.Q -> DigitalOutput_Q3.OUT
- AX_T_FF_Q4.Q -> DigitalOutput_Q4.OUT

Ereignisverbindungen:

- AX_X_TO_BOOL_2.CNF -> F_BOOL_TO_UINT_I2.REQ
- AX_X_TO_BOOL_3.CNF -> F_BOOL_TO_UINT_I3.REQ
- AX_X_TO_BOOL_4.CNF -> F_BOOL_TO_UINT_I4.REQ
- F_BOOL_TO_UINT_I4.CNF -> ADD_3.REQ
- F_BOOL_TO_UINT_I3.CNF -> ADD_3.REQ
- F_BOOL_TO_UINT_I2.CNF -> ADD_3.REQ
- E_DEMUX_4.EO1 -> AX_T_FF_Q1.CLK
- E_DEMUX_4.EO2 -> AX_T_FF_Q2.CLK
- E_DEMUX_4.EO3 -> AX_T_FF_Q3.CLK

- E_DEMUX_4.EO4 -> AX_T_FF_Q4.CLK
- DigitalInput_CLK_I1.IND -> E_DEMUX_4.EI

Datenverbindungen:

- AX_X_TO_BOOL_2.IN -> F_BOOL_TO_UINT_I2.IN
- AX_X_TO_BOOL_3.IN -> F_BOOL_TO_UINT_I3.IN
- AX_X_TO_BOOL_4.IN -> F_BOOL_TO_UINT_I4.IN
- F_BOOL_TO_UINT_I2.OUT -> ADD_3.IN1
- F_BOOL_TO_UINT_I3.OUT -> ADD_3.IN2
- F_BOOL_TO_UINT_I4.OUT -> ADD_3.IN3
- ADD_3.OUT -> E_DEMUX_4.K

Hinweise aus dem Modell

> Diese Taste schaltet ein oder aus Die Anzahl der restlichen Tasten bestimmt den Ausgang: Keine –> Q1 Eine –> Q2 Zwei –> Q3 Drei –> Q4

Programmablauf und Funktionsweise

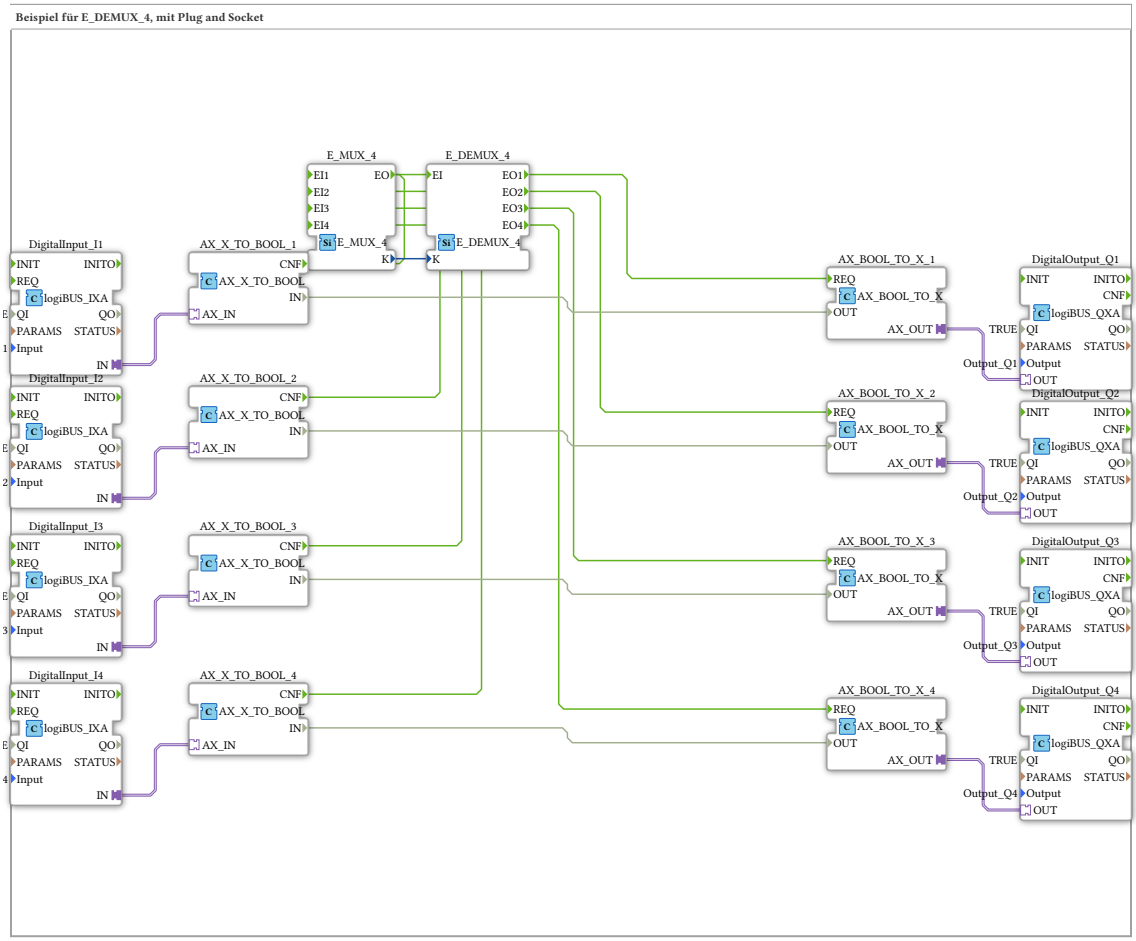
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_087a1_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.391 Uebung_087a2_AX: Beispiel für E_DEMUX_4, mit Plug and Socket

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_087a2_AX (Beispiel für E_DEMUX_4, mit Plug and Socket).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Beispiel für E_DEMUX_4, mit Plug and Socket**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_087a2_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
- **DigitalInput_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
- **DigitalOutput_Q2**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q2
- **DigitalInput_I2**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I2
- **DigitalOutput_Q3**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q3
- **DigitalInput_I3**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I3

- **DigitalOutput_Q4**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q4
- **DigitalInput_I4**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I4
- **E_DEMUX_4**: Instanz des Typs iec61499::events::E_DEMUX_4.
- **E_MUX_4**: Instanz des Typs iec61499::events::E_MUX_4.
- **AX_X_TO_BOOL_1**: Instanz des Typs adapter::conversion::unidirectional::AX_X_TO_BOOL.
- **AX_X_TO_BOOL_2**: Instanz des Typs adapter::conversion::unidirectional::AX_X_TO_BOOL.
- **AX_X_TO_BOOL_3**: Instanz des Typs adapter::conversion::unidirectional::AX_X_TO_BOOL.
- **AX_X_TO_BOOL_4**: Instanz des Typs adapter::conversion::unidirectional::AX_X_TO_BOOL.
- **AX_BOOL_TO_X_1**: Instanz des Typs adapter::conversion::unidirectional::AX_BOOL_TO_X.
- **AX_BOOL_TO_X_2**: Instanz des Typs adapter::conversion::unidirectional::AX_BOOL_TO_X.
- **AX_BOOL_TO_X_3**: Instanz des Typs adapter::conversion::unidirectional::AX_BOOL_TO_X.
- **AX_BOOL_TO_X_4**: Instanz des Typs adapter::conversion::unidirectional::AX_BOOL_TO_X.

Verbindungen und Schnittstellen

Adapterverbindungen:

- DigitalInput_I1.IN -> AX_X_TO_BOOL_1.AX_IN
- DigitalInput_I2.IN -> AX_X_TO_BOOL_2.AX_IN
- DigitalInput_I3.IN -> AX_X_TO_BOOL_3.AX_IN
- DigitalInput_I4.IN -> AX_X_TO_BOOL_4.AX_IN
- AX_BOOL_TO_X_1.AX_OUT -> DigitalOutput_Q1.OUT
- AX_BOOL_TO_X_2.AX_OUT -> DigitalOutput_Q2.OUT
- AX_BOOL_TO_X_3.AX_OUT -> DigitalOutput_Q3.OUT
- AX_BOOL_TO_X_4.AX_OUT -> DigitalOutput_Q4.OUT

Ereignisverbindungen:

- E_MUX_4.EO -> E_DEMUX_4.EI
- AX_X_TO_BOOL_1.CNF -> E_MUX_4.EI1
- AX_X_TO_BOOL_2.CNF -> E_MUX_4.EI2
- AX_X_TO_BOOL_3.CNF -> E_MUX_4.EI3
- AX_X_TO_BOOL_4.CNF -> E_MUX_4.EI4
- E_DEMUX_4.EO4 -> AX_BOOL_TO_X_4.REQ
- E_DEMUX_4.EO3 -> AX_BOOL_TO_X_3.REQ
- E_DEMUX_4.EO2 -> AX_BOOL_TO_X_2.REQ
- E_DEMUX_4.EO1 -> AX_BOOL_TO_X_1.REQ

Datenverbindungen:

- E_MUX_4.K -> E_DEMUX_4.K
- AX_X_TO_BOOL_1.IN -> AX_BOOL_TO_X_1.OUT
- AX_X_TO_BOOL_2.IN -> AX_BOOL_TO_X_2.OUT
- AX_X_TO_BOOL_3.IN -> AX_BOOL_TO_X_3.OUT
- AX_X_TO_BOOL_4.IN -> AX_BOOL_TO_X_4.OUT

Programmablauf und Funktionsweise

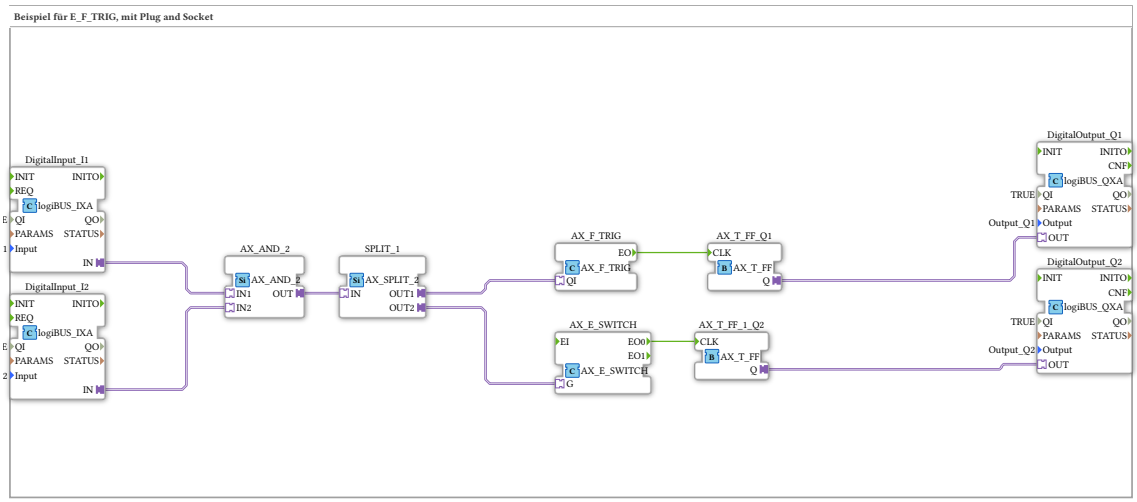
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_087a2_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.392 Uebung_088_AX: Beispiel für E_F_TRIG, mit Plug and Socket

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_088_AX (Beispiel für E_F_TRIG, mit Plug and Socket).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Beispiel für E_F_TRIG, mit Plug and Socket**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_088_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
- **DigitalInput_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
- **AX_AND_2**: Instanz des Typs adapter::booleanOperators::AX_AND_2.
- **DigitalInput_I2**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I2
- **AX_F_TRIG**: Instanz des Typs adapter::events::unidirectional::AX_F_TRIG.
- **AX_T_FF_Q1**: Instanz des Typs adapter::events::unidirectional::AX_T_FF.
- **DigitalOutput_Q2**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q2
- **AX_T_FF_1_Q2**: Instanz des Typs adapter::events::unidirectional::AX_T_FF.
- **AX_E_SWITCH**: Instanz des Typs adapter::events::unidirectional::AX_E_SWITCH.
- **SPLIT_1**: Instanz des Typs adapter::events::unidirectional::AX_SPLIT_2.

Verbindungen und Schnittstellen

Adapterverbindungen:

- DigitalInput_I1.IN -> AX_AND_2.IN1
- DigitalInput_I2.IN -> AX_AND_2.IN2
- AX_T_FF_Q1.Q -> DigitalOutput_Q1.OUT
- AX_T_FF_1_Q2.Q -> DigitalOutput_Q2.OUT
- AX_AND_2.OUT -> SPLIT_1.IN
- SPLIT_1.OUT1 -> AX_F_TRIG.QI
- SPLIT_1.OUT2 -> AX_E_SWITCH.G

Ereignisverbindungen:

- AX_F_TRIG.EO -> AX_T_FF_Q1.CLK
- AX_E_SWITCH.EO0 -> AX_T_FF_1_Q2.CLK

Hinweise aus dem Modell

F_TRIG schaltet nur wenn wirklich fallende Flanke E_SWITCH schaltet auch, wenn anderweitig ein Event kommt.

Programmablauf und Funktionsweise

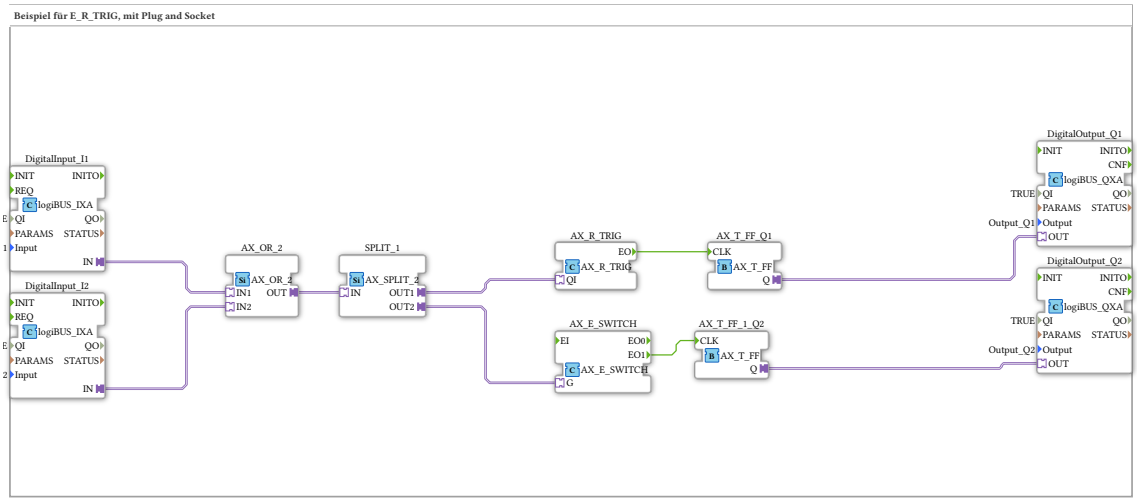
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_088_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.393 Uebung_089_AX: Beispiel für E_R_TRIG, mit Plug and Socket

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_089_AX (Beispiel für E_R_TRIG, mit Plug and Socket).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Beispiel für E_R_TRIG, mit Plug and Socket**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_089_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
- **DigitalInput_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1

- **AX_OR_2**: Instanz des Typs adapter::booleanOperators::AX_OR_2.
- **SPLIT_1**: Instanz des Typs adapter::events::unidirectional::AX_SPLIT_2.
- **DigitalInput_I2**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I2
- **AX_R_TRIG**: Instanz des Typs adapter::events::unidirectional::AX_R_TRIG.
- **AX_T_FF_Q1**: Instanz des Typs adapter::events::unidirectional::AX_T_FF.
- **DigitalOutput_Q2**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q2
- **AX_T_FF_1_Q2**: Instanz des Typs adapter::events::unidirectional::AX_T_FF.
- **AX_E_SWITCH**: Instanz des Typs adapter::events::unidirectional::AX_E_SWITCH.

Verbindungen und Schnittstellen

Adapterverbindungen:

- DigitalInput_I1.IN -> AX_OR_2.IN1
- DigitalInput_I2.IN -> AX_OR_2.IN2
- AX_T_FF_Q1.Q -> DigitalOutput_Q1.OUT
- AX_T_FF_1_Q2.Q -> DigitalOutput_Q2.OUT
- AX_OR_2.OUT -> SPLIT_1.IN
- SPLIT_1.OUT1 -> AX_R_TRIG.QI
- SPLIT_1.OUT2 -> AX_E_SWITCH.G

Ereignisverbindungen:

- AX_R_TRIG.EO -> AX_T_FF_Q1.CLK
- AX_E_SWITCH.EO1 -> AX_T_FF_1_Q2.CLK

Hinweise aus dem Modell

R_TRIG schaltet nur wenn wirklich steigende Flanke E_SWITCH schaltet auch, wenn anderweitig ein Event kommt.

Programmablauf und Funktionsweise

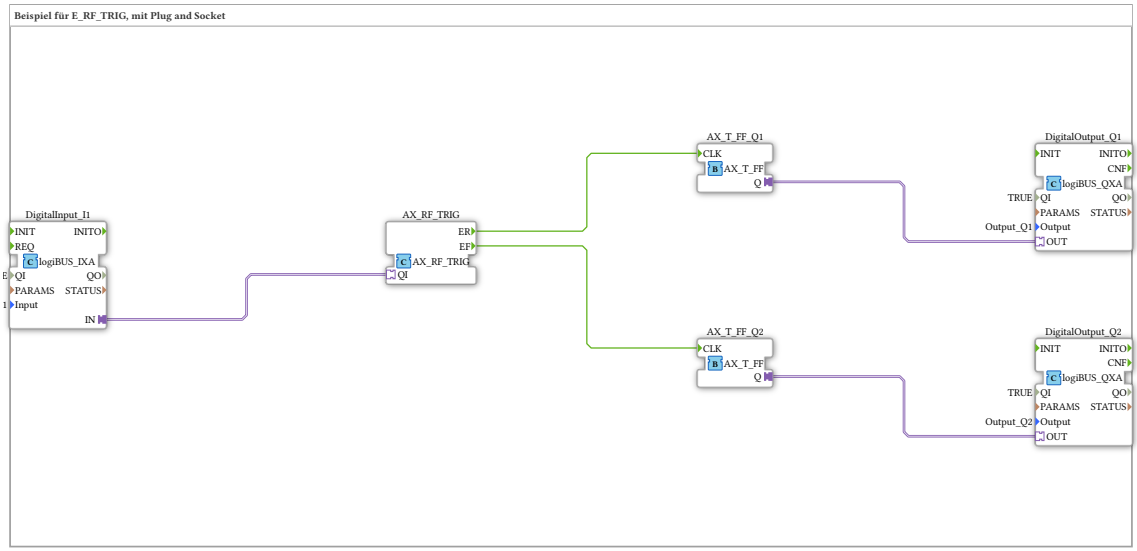
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_089_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.394 Uebung_089a_AX: Beispiel für E_RF_TRIG, mit Plug and Socket

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_089a_AX (Beispiel für E_RF_TRIG, mit Plug and Socket).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Beispiel für E_RF_TRIG, mit Plug and Socket**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_089a_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
- **AX_RF_TRIG**: Instanz des Typs adapter::events::unidirectional::AX_RF_TRIG.
- **AX_T_FF_Q1**: Instanz des Typs adapter::events::unidirectional::AX_T_FF.
- **AX_T_FF_Q2**: Instanz des Typs adapter::events::unidirectional::AX_T_FF.
- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
- **DigitalOutput_Q2**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q2

Verbindungen und Schnittstellen

Adapterverbindungen:

- DigitalInput_I1.IN -> AX_RF_TRIG.QI
- AX_T_FF_Q1.Q -> DigitalOutput_Q1.OUT
- AX_T_FF_Q2.Q -> DigitalOutput_Q2.OUT

Ereignisverbindungen:

- AX_RF_TRIG.ER -> AX_T_FF_Q1.CLK
- AX_RF_TRIG.EF -> AX_T_FF_Q2.CLK

Hinweise aus dem Modell

> Ein einziger Eingang, EIN Signal - AX_RF_TRIG liefert beide Flanken gleichzeitig (ER=steigend, EF=fallend). Q1 togglet bei steigender, Q2 bei fallender Flanke.

Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_089a_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.395 Uebung_090a1_AX: Beispiel für AX_MUX_2

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_090a1_AX`.

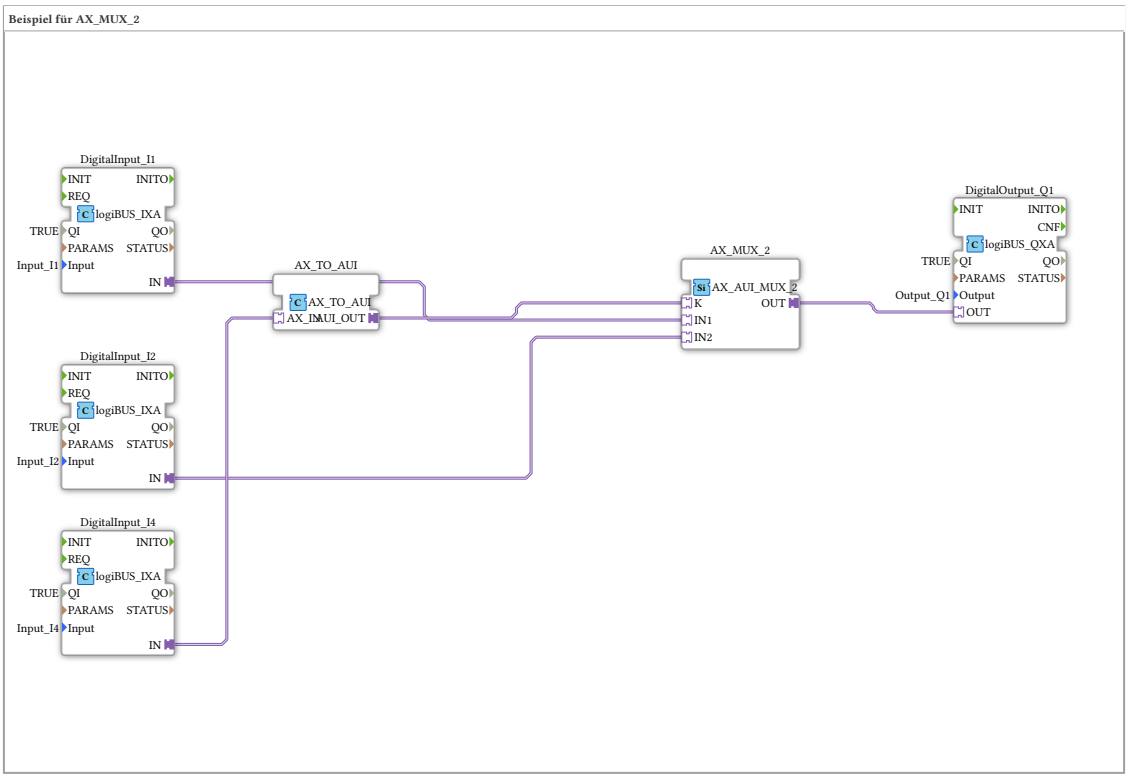
Ziel der Übung

Auswahl eines Signals aus mehreren Quellen (Umschalter).

Beschreibung und Komponenten

Die Subapplikation `Uebung_090a1_AX.SUB` verwendet einen `AX_MUX_2` Baustein.

Funktionsbausteine (FBs)



- **`I1` & `I2`**: Die beiden Signalquellen.
- **`I4`**: Der Wahlschalter (Selector).
- **`F_MUX_2`**: Der Multiplexer.
- **`F_BOOL_TO_UINT`**: Hilfsbaustein zur Konvertierung.

Funktionsweise

Der Multiplexer erwartet am Eingang `K` eine Ganzzahl (UINT), um zu entscheiden, welchen Eingang er durchschaltet. Da `I4` ein boolesches Signal liefert, wird dieses konvertiert:

- `I4 = FALSE` -> `K = 0` -> `MUX` schaltet `IN1` (`I1`) auf den Ausgang.
- `I4 = TRUE` -> `K = 1` -> `MUX` schaltet `IN2` (`I2`) auf den Ausgang.

Der Ausgang `Q1` folgt also entweder `I1` oder `I2`, abhängig von der Stellung von `I4`.

Anwendungsbeispiel

Hand/Automatik-Umschaltung:

- `I1`: Signal aus der Automatik-Steuerung.
- `I2`: Signal vom Hand-Taster.
- `I4`: Schlüsselschalter „Hand/Auto“. Der Ausgang (`Q1`) wird je nach Betriebsart von der Automatik oder manuell gesteuert.

2.1.2.396 Uebung_090a2_AX: Beispiel für AX_MUX_3



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_090a2_AX`.

Ziel der Übung

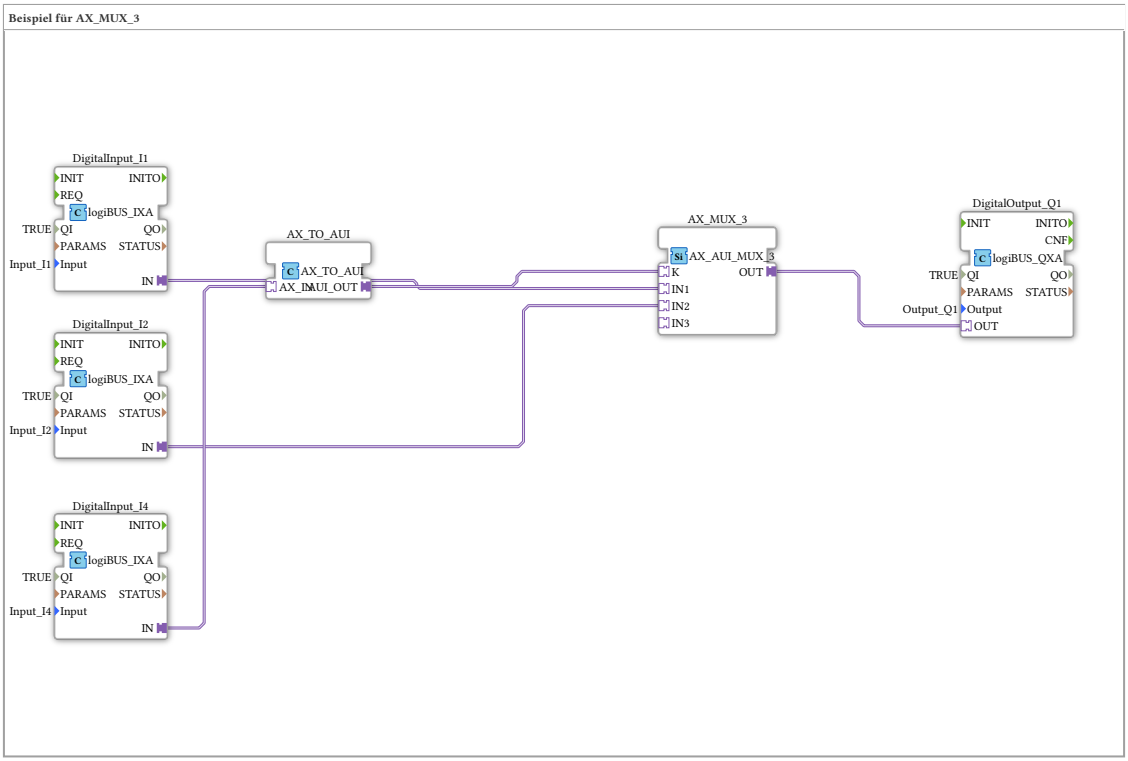
Erweiterung des Multiplexers.

Beschreibung

Strukturell identisch zu `Uebung_090a1_AX`, jedoch mit einem `AX_MUX_3`.

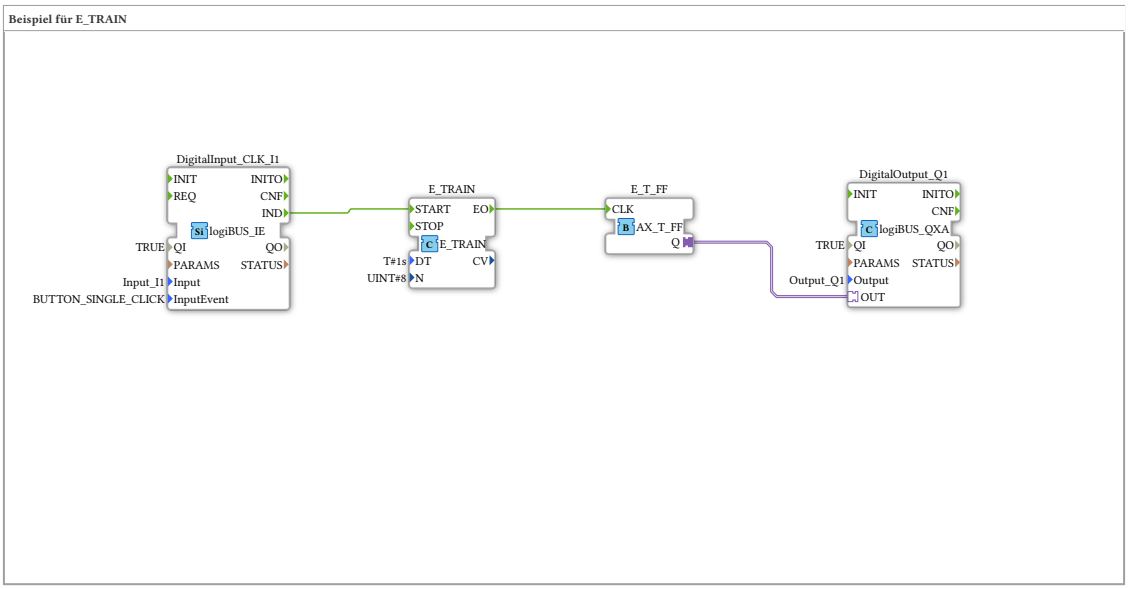
Funktionsweise

Da als Selektor weiterhin nur ein boolescher Eingang (`I4`) verwendet wird, können nur die ersten beiden Eingänge (`IN1` bei K=0, `IN2` bei K=1) ausgewählt werden. Der dritte Eingang (`IN3`, Index 2) ist in dieser Konstellation nicht erreichbar. Um alle drei zu nutzen, bräuchte man einen Integer-Eingang oder zwei Bit-Eingänge.



2.1.2.397 Uebung_091_AX: Beispiel für E_TRAIN

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_091_AX (Beispiel für E_TRAIN).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Beispiel für E_TRAIN**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_091_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.

  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
- **E_TRAIN**: Instanz des Typs iec61499::events::E_TRAIN.
  - Parameter DT = T#1s
  - Parameter N = 8
- **DigitalInput_CLK_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **E_T_FF**: Instanz des Typs adapter::events::unidirectional::AX_T_FF.

Verbindungen und Schnittstellen

Adapterverbindungen:

- E_T_FF.Q -> DigitalOutput_Q1.OUT

Ereignisverbindungen:

- DigitalInput_CLK_I1.IND -> E_TRAIN.START
- E_TRAIN.EO -> E_T_FF.CLK

Hinweise aus dem Modell

E_TRAIN wird 8 Events ausgeben, das erste 1s nach dem Click, das letzte 8s nach dem Click

---

Programmablauf und Funktionsweise

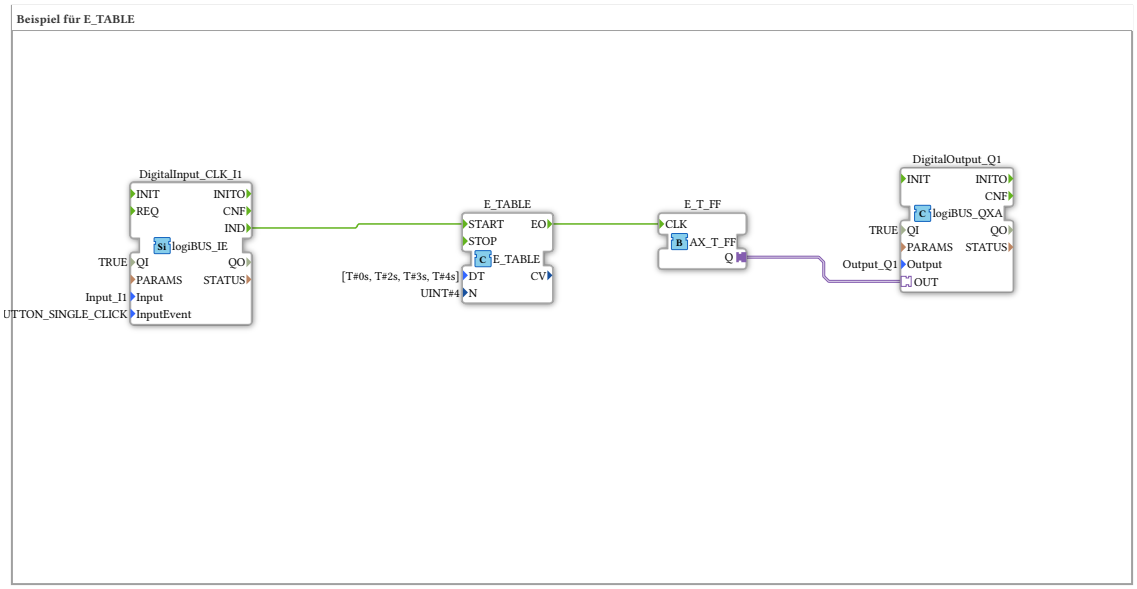
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_091_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.398 Uebung_093_AX: Beispiel für E_TABLE

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_093_AX (Beispiel für E_TABLE).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Beispiel für E_TABLE**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_093_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
- **E_TABLE**: Instanz des Typs iec61499::events::E_TABLE.
  - Parameter DT = [T#0s, T#2s, T#3s, T#4s]
  - Parameter N = 4
- **DigitalInput_CLK_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **E_T_FF**: Instanz des Typs adapter::events::unidirectional::AX_T_FF.

Verbindungen und Schnittstellen

Adapterverbindungen:

- E_T_FF.Q -> DigitalOutput_Q1.OUT

Ereignisverbindungen:

- DigitalInput_CLK_I1.IND -> E_TABLE.START
- E_TABLE.EO -> E_T_FF.CLK

Hinweise aus dem Modell

> E_TABLE wird 4 Events ausgeben, das erste sofort nach dem Click, das letzte 9s nach dem Click [T#0s, T#2s, T#3s, T#4s]

Programmablauf und Funktionsweise

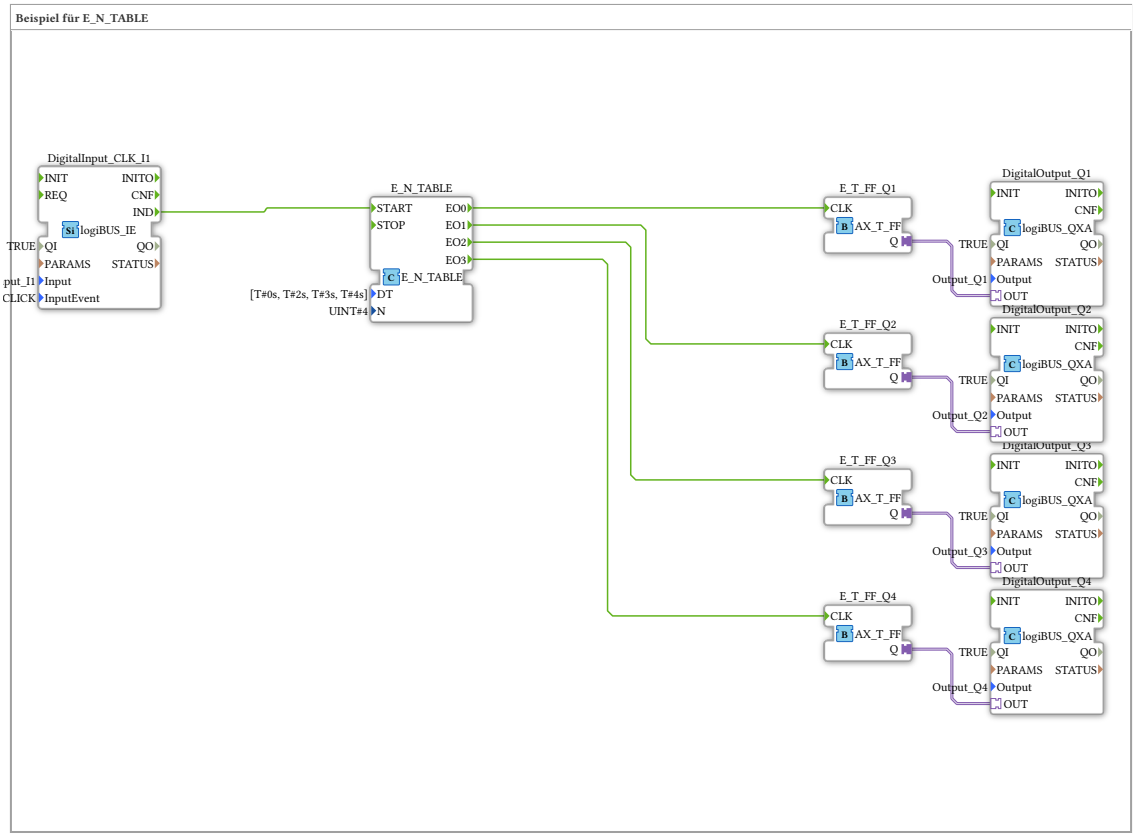
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_093_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.399 Uebung_093b_AX: Beispiel für E_N_TABLE

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_093b_AX (Beispiel für E_N_TABLE).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Beispiel für E_N_TABLE**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_093b_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **E_N_TABLE**: Instanz des Typs iec61499::events::E_N_TABLE.
  - Parameter DT = [T#0s, T#2s, T#3s, T#4s]
  - Parameter N = 4
- **DigitalInput_CLK_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **E_T_FF_Q2**: Instanz des Typs adapter::events::unidirectional::AX_T_FF.
- **E_T_FF_Q4**: Instanz des Typs adapter::events::unidirectional::AX_T_FF.
- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
- **E_T_FF_Q3**: Instanz des Typs adapter::events::unidirectional::AX_T_FF.
- **DigitalOutput_Q3**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q3
- **E_T_FF_Q1**: Instanz des Typs adapter::events::unidirectional::AX_T_FF.
- **DigitalOutput_Q2**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q2
- **DigitalOutput_Q4**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE

- Parameter Output = Output_Q4

Verbindungen und Schnittstellen

Adapterverbindungen:

- E_T_FF_Q1.Q -> DigitalOutput_Q1.OUT
- E_T_FF_Q4.Q -> DigitalOutput_Q4.OUT
- E_T_FF_Q2.Q -> DigitalOutput_Q2.OUT
- E_T_FF_Q3.Q -> DigitalOutput_Q3.OUT

Ereignisverbindungen:

- DigitalInput_CLK_I1.IND -> E_N_TABLE.START
- E_N_TABLE.EO0 -> E_T_FF_Q1.CLK
- E_N_TABLE.EO1 -> E_T_FF_Q2.CLK
- E_N_TABLE.EO2 -> E_T_FF_Q3.CLK
- E_N_TABLE.EO3 -> E_T_FF_Q4.CLK

Hinweise aus dem Modell

> E_TABLE wird 4 Events ausgeben, das erste sofort nach dem Click, das letzte 9s nach dem Click [T#0s, T#2s, T#3s, T#4s]

Programmablauf und Funktionsweise

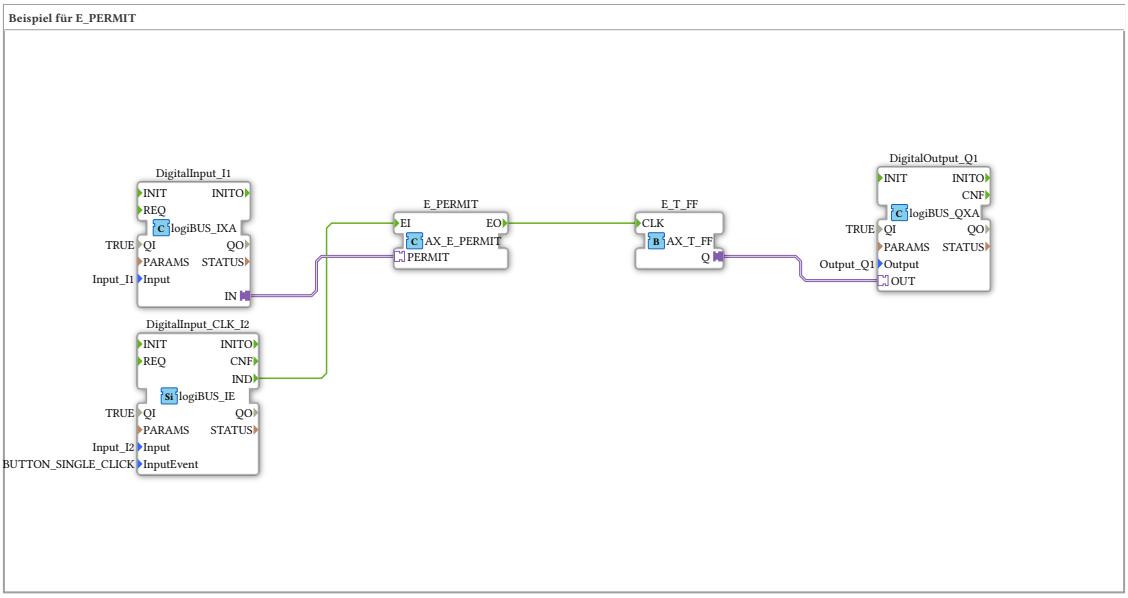
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_093b_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.400 Uebung_094_AX: Beispiel für E_PERMIT

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_094_AX (Beispiel für E_PERMIT).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Beispiel für E_PERMIT**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_094_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
- **DigitalInput_CLK_I2**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I2
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **E_PERMIT**: Instanz des Typs adapter::events::unidirectional::AX_E_PERMIT.
- **E_T_FF**: Instanz des Typs adapter::events::unidirectional::AX_T_FF.
- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1

Verbindungen und Schnittstellen

Adapterverbindungen:

- DigitalInput_I1.IN -> E_PERMIT.PERMIT
- E_T_FF.Q -> DigitalOutput_Q1.OUT

Ereignisverbindungen:

- DigitalInput_CLK_I2.IND -> E_PERMIT.EI
- E_PERMIT.EO -> E_T_FF.CLK

Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_094_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.401 Uebung_094a_AX: Beispiel für QI anstelle Permit



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_094a_AX`.

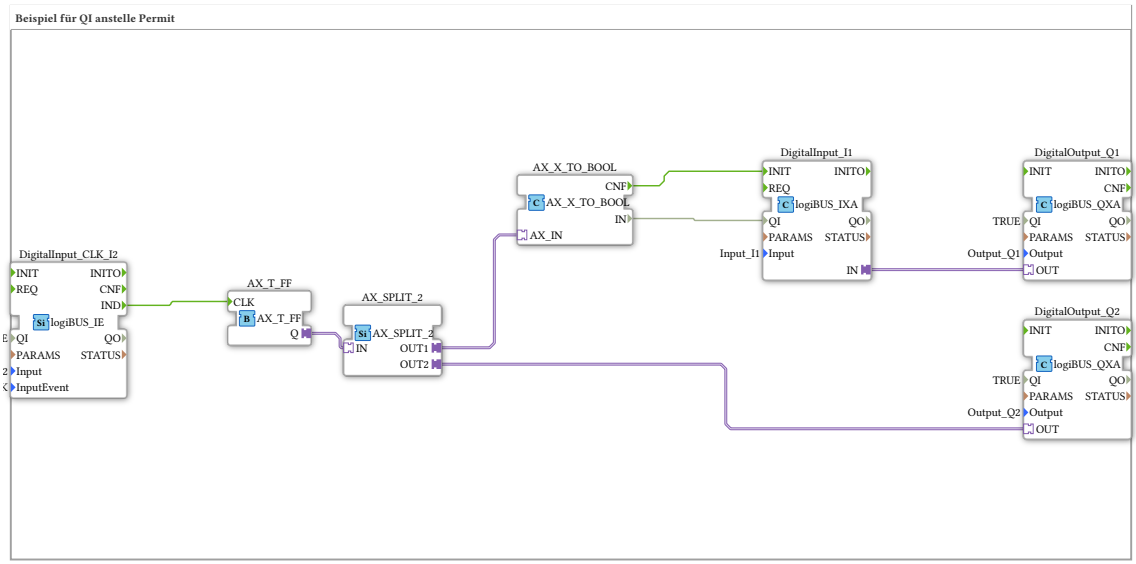
Ziel der Übung

Nutzung des `QI` (Qualifier Input) Parameters zur Laufzeit-Steuerung von Funktionsbausteinen.

Beschreibung und Komponenten

Die Subapplikation `Uebung_094a_AX.SUB` schaltet einen Eingangspfad aktiv oder inaktiv.

Funktionsbausteine (FBs)



- **`DigitalInput_CLK_I2`**: Toggelt über ein Flip-Flop den Zustand „Aktiv/Inaktiv“.
- **`DigitalInput_I1`**: Der eigentliche Signaleingang. Sein `QI` Parameter ist variabel beschaltet.
- **`DigitalOutput_Q1`**: Hängt an `I1`.
- **`DigitalOutput_Q2`**: Zeigt den Status „Ist Aktiv“ an.

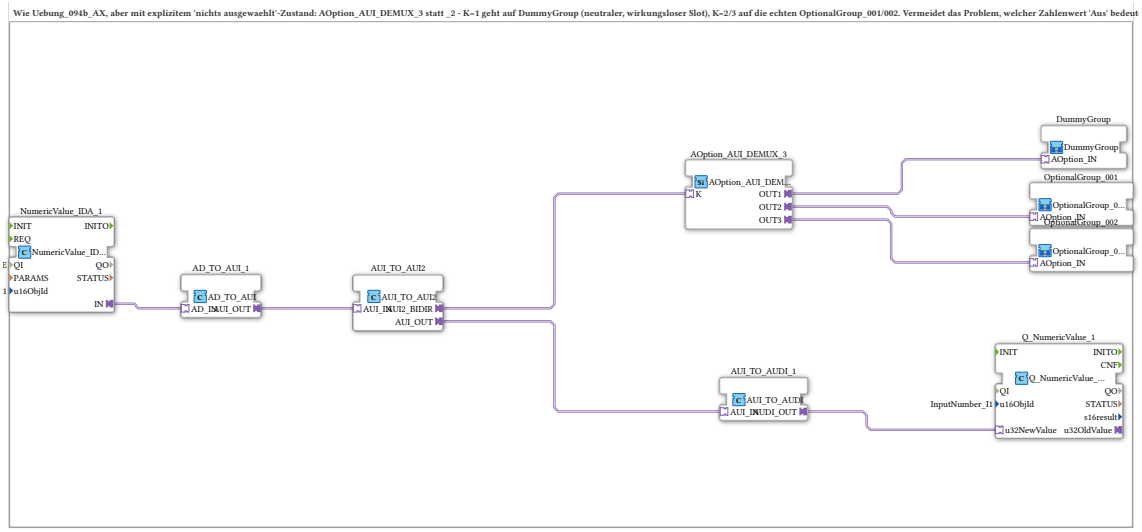
Funktionsweise

1. Drückt man `I2`, schaltet `Q2` ein (System aktiviert).
2. Gleichzeitig wird `QI` von `DigitalInput_I1` auf TRUE gesetzt.
3. Jetzt funktioniert `I1`: Wenn man `I1` drückt, schaltet `Q1`.
4. Drückt man `I2` erneut, schaltet `Q2` aus (System deaktiviert).
5. `QI` von `I1` wird FALSE.
6. Der Baustein `I1` stellt seine Arbeit ein. Änderungen am physischen Eingang `I1` werden nicht mehr an `Q1` weitergeleitet.

Anwendungsbeispiel

Wartungsmodus: Teile der Sensorik werden softwareseitig „abgeklemmt“, damit bei Arbeiten an der Maschine keine Fehlalarme ausgelöst werden.

2.1.2.402 Uebung_094b2_AX: Optionsauswahl per VT-Eingabefeld mit explizitem „nichts ausgewählt“-Zustand



• – ○ • – ○ • – ○ •

Einleitung

Die Übung entspricht `Uebung_094b_AX`, verwendet aber `AOption_AUI_DEMUX_3` statt `AOption_AUI_DEMUX_2`. Der Wert `K = 1` geht auf eine neutrale `DummyGroup` ohne Wirkung, `K = 2` und `K = 3` aktivieren die echten Gruppen `OptionalGroup_001` bzw. `OptionalGroup_002`. Damit entfällt die Frage, welcher Zahlenwert „Aus“ bedeuten soll: Es wird einfach ein eigener Platzhalter-Slot im DEMUX reserviert. Die Lese-/Echo-Kette zum VT-Eingabefeld ist hier noch inline verdrahtet.

Verwendete Funktionsbausteine (FBs)

- **NumericValue_IDA_1**: liest das VT-Eingabefeld
  - **Typ**: `isobus::UT::io::NumericValue::NumericValue_IDA`
  - **Parameter**: `QI = TRUE`, `u16ObjId = InputNumber_I1`
  - **Erklärung**: Liefert den Wert des VT-Zahlenfelds `InputNumber_I1` über den Adapter `IN` (AD).
- **AD_TO_AUI_1**: Adapterumsetzung AD nach AUI
  - **Typ**: `adapter::conversion::unidirectional::AD_TO_AUI`
  - **Parameter**: keine
  - **Erklärung**: Wandelt den DWORD-Wert in einen UINT-Wert (AUI).
- **AUI_TO_AUI2**: Brücke zum bidirektionalen AUI2-Adapter
  - **Typ**: `adapter::conversion::bidirectional::AUI_TO_AUI2`
  - **Parameter**: keine
  - **Erklärung**: Reicht den Wert über `AUI2_BIDIR` an den Consumer (`K` des DEMUX) weiter und nimmt von dort den tatsächlich übernommenen Wert als Echo zurück (`AUI_OUT`).
- **AUI_TO_AUDI_1**: Adapterumsetzung AUI nach AUDI
  - **Typ**: `adapter::conversion::unidirectional::AUI_TO_AUDI`
  - **Parameter**: keine
  - **Erklärung**: Wandelt das Echo wieder in einen UDINT-Wert für das VT.
- **Q_NumericValue_1**: schreibt den Wert ins VT
  - **Typ**: `isobus::UT::Q::Q_NumericValue_AUDI`
  - **Parameter**: `u16ObjId = InputNumber_I1`
  - **Erklärung**: Schreibt den zurückgemeldeten Wert (`u32NewValue`) in das VT-Feld `InputNumber_I1`.
- **AOption_AUI_DEMUX_3**: Optionsauswahl mit drei Ausgängen
  - **Typ**: `Option::selection::AOption_AUI_DEMUX_3`
  - **Parameter**: keine
  - **Erklärung**: Aktiviert abhängig vom Index `K` genau einen der Ausgänge `OUT1`, `OUT2`, `OUT3`.
- **OptionalGroup_001**, **OptionalGroup_002**: echte Optionsgruppen
  - **Typ**: `MyLib_AX_Option::sys::OptionalGroup_001` bzw. `MyLib_AX_Option::sys::OptionalGroup_002` (SubApp-Instanzen)

  - **Parameter**: keine
  - **Erklärung**: Gruppe 1 enthält den Digitaleingang `I1` und den Digitalausgang `Q1`, Gruppe 2 `I2` und `Q2`. Über den Options-Adapter wird die ganze Gruppe aktiviert bzw. deaktiviert.
- **DummyGroup**: neutraler Platzhalter-Slot
  - **Typ**: `MyLib_AX_Option::sys::DummyGroup` (SubApp-Instanz)
  - **Parameter**: keine
  - **Erklärung**: Hat keine reale I/O und keine Wirkung; quittiert die Initialisierung selbst. Dient als Zustand „nichts ausgewählt“.

Programmablauf und Verbindungen

1. `NumericValue_IDA_1.IN` ⟶ `AD_TO_AUI_1.AD_IN` ⟶ `AUI_TO_AUI2.AUI_IN`: Der Wert des Eingabefelds wird nach AUI und dann in den bidirektionalen AUI2-Adapter überführt.
2. `AUI_TO_AUI2.AUI2_BIDIR` ⟶ `AOption_AUI_DEMUX_3.K`: Der Wert wählt den aktiven Ausgang des DEMUX.
3. `AOption_AUI_DEMUX_3.OUT1` ⟶ `DummyGroup.AOption_IN`, `OUT2` ⟶ `OptionalGroup_001.AOption_IN`, `OUT3` ⟶ `OptionalGroup_002.AOption_IN`.
4. Echo-Pfad: `AUI_TO_AUI2.AUI_OUT` ⟶ `AUI_TO_AUDI_1.AUI_IN` ⟶ `Q_NumericValue_1.u32NewValue`: Der vom DEMUX übernommene Wert wird in das VT-Feld zurückgeschrieben.

Verhalten: Im Eingabefeld `InputNumber_I1` ergibt 1 den Zustand „nichts ausgewählt“ (DummyGroup), 2 aktiviert Gruppe 1 und 3 aktiviert Gruppe 2.

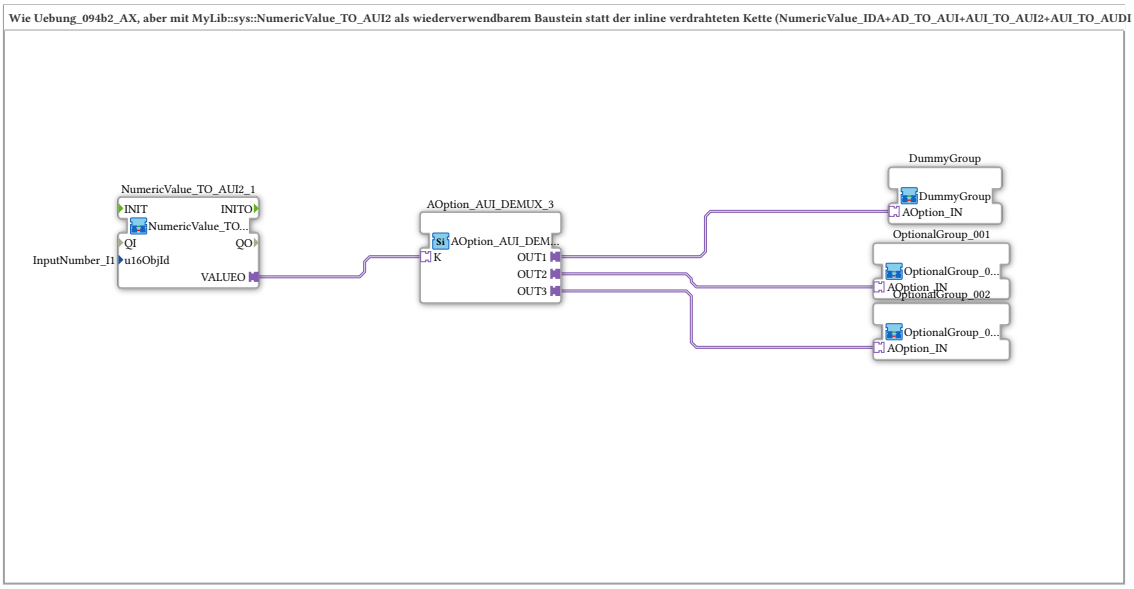
Zusammenfassung

Durch den zusätzlichen Platzhalter-Slot im `AOption_AUI_DEMUX_3` ist ein eindeutiger „nichts ausgewählt“-Zustand ohne besondere Zahlenkonvention möglich. Die inline verdrahtete Lese-/Echo-Kette wird in `Uebung_094b3_AX` in einen wiederverwendbaren Baustein ausgelagert.

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

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

2.1.2.403 Uebung_094b3_AX: Optionsauswahl mit wiederverwendbarem Baustein NumericValue_TO_AUI2



• – ○ • – ○ • – ○ •

Einleitung

Die Übung entspricht `Uebung_094b2_AX`, verwendet aber den wiederverwendbaren Baustein `MyLib::sys::NumericValue_TO_AUI2` statt der inline verdrahteten Kette aus `NumericValue_IDA`,

AD_TO_AUI, AUI_TO_AUI2, AUI_TO_AUDI und Q_NumericValue_AUDI. Die Objekt-ID des VT-Felds wird als SubApp-Parameter übergeben.

Verwendete Funktionsbausteine (FBs)

- **NumericValue_TO_AUI2_1**: liest das VT-Eingabefeld und stellt es als bidirektionalen AUI2-Plug bereit
  - **Typ**: MyLib::sys::NumericValue_TO_AUI2 (SubApp-Instanz)
  - **Parameter**: u16ObjId = InputNumber_I1
  - **Erklärung**: Liest das VT-Eingabefeld und reicht den Wert über VALUEO an einen Consumer weiter (hier K des DEMUX). Der Baustein speichert nicht selbst: Der Consumer meldet den tatsächlich übernommenen Wert über denselben Plug zurück, das Echo wird auf das VT-Feld zurückgeschrieben.
- **AOption_AUI_DEMUX_3**: Optionsauswahl mit drei Ausgängen
  - **Typ**: Option::selection::AOption_AUI_DEMUX_3
  - **Parameter**: keine
  - **Erklärung**: Aktiviert abhängig vom Index K genau einen der Ausgänge OUT1, OUT2, OUT3.
- **OptionalGroup_001**, **OptionalGroup_002**: echte Optionsgruppen
  - **Typ**: MyLib_AX_Option::sys::OptionalGroup_001 bzw. MyLib_AX_Option::sys::OptionalGroup_002 (SubApp-Instanzen)
  - **Parameter**: keine
  - **Erklärung**: Gruppe 1 enthält I1/Q1, Gruppe 2 enthält I2/Q2; die ganze Gruppe wird über den Options-Adapter aktiviert bzw. deaktiviert.
- **DummyGroup**: neutraler Platzhalter-Slot
  - **Typ**: MyLib_AX_Option::sys::DummyGroup (SubApp-Instanz)
  - **Parameter**: keine
  - **Erklärung**: Ohne reale I/O und ohne Wirkung; Zustand „nichts ausgewählt“.

Programmablauf und Verbindungen

1. NumericValue_TO_AUI2_1.VALUEO ⟶ AOption_AUI_DEMUX_3.K: Der Wert des Eingabefelds wählt den aktiven Ausgang.
2. AOption_AUI_DEMUX_3.OUT1 ⟶ DummyGroup.AOption_IN, OUT2 ⟶ OptionalGroup_001.AOption_IN, OUT3 ⟶ OptionalGroup_002.AOption_IN.

Verhalten: Wie in Uebung_094b2_AX: 1 = nichts ausgewählt, 2 = Gruppe 1, 3 = Gruppe 2.

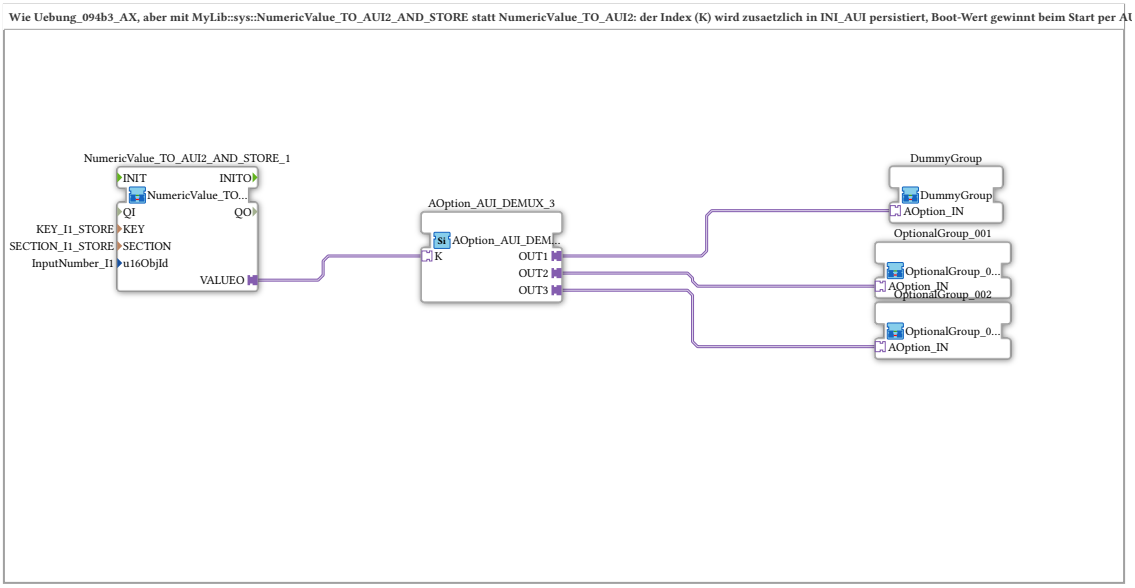
Zusammenfassung

Die Lese-/Echo-Kette ist in MyLib::sys::NumericValue_TO_AUI2 gekapselt; das Netzwerk der Übung schrumpft auf vier Instanzen und zwei Verbindungsarten, bei gleichem Verhalten wie Uebung_094b2_AX.

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

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

2.1.2.404 Uebung_094b3b_AX: Optionsauswahl mit INI-persistiertem Index (NumericValue_TO_AUI2_AND_STORE)



• – ◦ • – ◦ • – ◦ •

Einleitung

Die Übung entspricht `Uebung_094b3_AX`, verwendet aber `MyLib::sys::NumericValue_TO_AUI2_AND_STORE` statt `NumericValue_TO_AUI2`. Der Plug `VALUEO` ist derselbe, zusätzlich wird der Index `K` in einer INI-Datei persistiert. Beim Start gewinnt der gespeicherte Wert per `AUI_LAST_2` gegen die Live-Lesung des VT-Felds. Die Konstanten `KEY_I1_STORE` und `SECTION_I1_STORE` sind bereits aus `Uebung_012c_AUDI` bekannt.

Verwendete Funktionsbausteine (FBs)

- **NumericValue_TO_AUI2_AND_STORE_1**: VT-Eingabefeld lesen, als AUI2-Plug bereitstellen und in INI speichern
  - **Typ**: `MyLib::sys::NumericValue_TO_AUI2_AND_STORE` (SubApp-Instanz)
  - **Parameter**: `KEY = KEY_I1_STORE`, `SECTION = SECTION_I1_STORE`, `u16ObjId = InputNumber_I1`
  - **Erklärung**: Wie `NumericValue_TO_AUI2`, mit zusätzlicher INI-Persistenz: Beim Boot gewinnt der aus der INI geladene Wert (per `AUI_LAST_2` gegen die Live-VT-Lesung gemergt), danach jede Live-Änderung.
- **AOption_AUI_DEMUX_3**: Optionsauswahl mit drei Ausgängen
  - **Typ**: `Option::selection::AOption_AUI_DEMUX_3`
  - **Parameter**: keine
  - **Erklärung**: Aktiviert abhängig vom Index `K` genau einen der Ausgänge `OUT1`, `OUT2`, `OUT3`.
- **OptionalGroup_001**, **OptionalGroup_002**: echte Optionsgruppen
  - **Typ**: `MyLib_AX_Option::sys::OptionalGroup_001` bzw. `MyLib_AX_Option::sys::OptionalGroup_002` (SubApp-Instanzen)
  - **Parameter**: keine
  - **Erklärung**: Gruppe 1 enthält `I1`/`Q1`, Gruppe 2 enthält `I2`/`Q2`.
- **DummyGroup**: neutraler Platzhalter-Slot
  - **Typ**: `MyLib_AX_Option::sys::DummyGroup` (SubApp-Instanz)
  - **Parameter**: keine
  - **Erklärung**: Ohne reale I/O und ohne Wirkung; Zustand „nichts ausgewählt“.

Programmablauf und Verbindungen

1. `NumericValue_TO_AUI2_AND_STORE_1.VALUEO` ⟶ `AOption_AUI_DEMUX_3.K`: Der (ggf. aus der INI geladene) Index wählt den aktiven Ausgang.
2. `AOption_AUI_DEMUX_3.OUT1` ⟶ `DummyGroup.AOption_IN`, `OUT2` ⟶ `OptionalGroup_001.AOption_IN`, `OUT3` ⟶ `OptionalGroup_002.AOption_IN`.

Verhalten: Wie in `Uebung_094b3_AX` (1 = nichts ausgewählt, 2 = Gruppe 1, 3 = Gruppe 2), zusätzlich bleibt die Auswahl über einen Neustart erhalten.

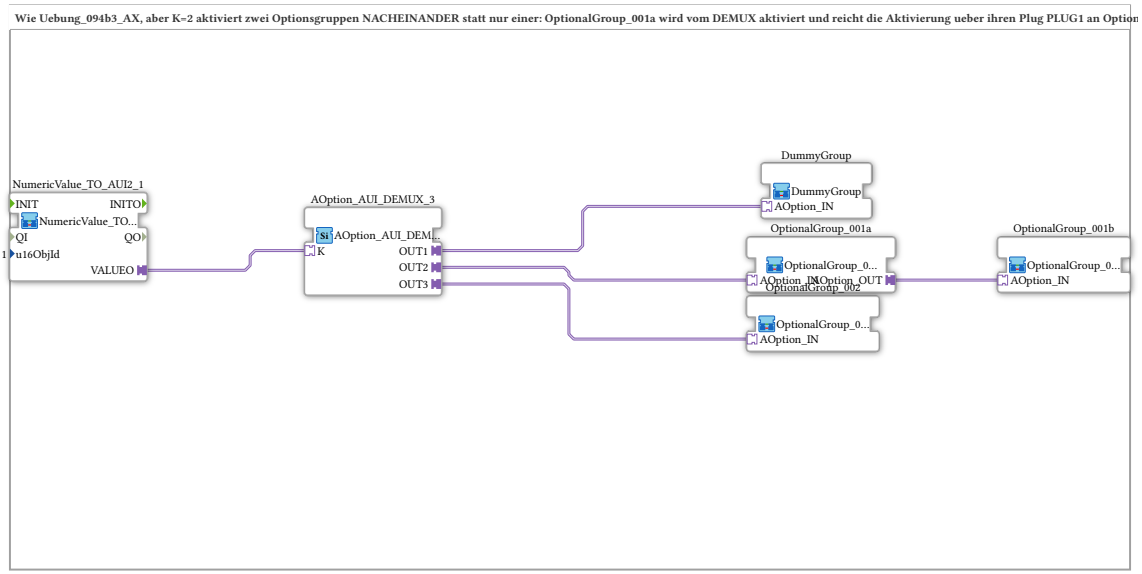
Zusammenfassung

Der Austausch des Bausteins durch die `_AND_STORE`-Variante macht die Optionsauswahl neustartfest, ohne dass sich das restliche Netzwerk ändert.

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

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

2.1.2.405 Uebung_094b3c_AX: Optionsauswahl mit zwei nacheinander aktivierten Optionsgruppen (Daisy-Chain)



• – ○ • – ○ • – ○ •

Einleitung

Die Übung entspricht `Uebung_094b3_AX`, aber der Index `K = 2` aktiviert zwei Optionsgruppen NACHEINANDER statt nur einer: `OptionalGroup_001a` wird vom DEMUX aktiviert und reicht die Aktivierung über ihren Plug `AOption_OUT` an `OptionalGroup_001b` weiter. Der DEMUX bekommt seine Bestätigung erst, wenn beide Gruppen initialisiert sind. Die Übung zeigt, wie man über die Plug/Socket-Kette mehrere Gruppen sequenziell (nacheinander, nicht gleichzeitig) hochfahren kann, ohne das Busy-Gating des DEMUX zu verletzen.

Verwendete Funktionsbausteine (FBs)

- **NumericValue_TO_AUI2_1**: liest das VT-Eingabefeld und stellt es als bidirektionalen AUI2-Plug bereit
  - **Typ**: `MyLib::sys::NumericValue_TO_AUI2` (SubApp-Instanz)
  - **Parameter**: `u16ObjId = InputNumber_I1`
  - **Erklärung**: Liest das VT-Eingabefeld und reicht den Wert über `VALUEO` an den DEMUX weiter. Der Consumer meldet den übernommenen Wert über denselben Plug zurück, das Echo wird auf das VT-Feld zurückgeschrieben.
- **AOption_AUI_DEMUX_3**: Optionsauswahl mit drei Ausgängen
  - **Typ**: `Option::selection::AOption_AUI_DEMUX_3`
  - **Parameter**: keine
  - **Erklärung**: Aktiviert abhängig vom Index `K` genau einen der Ausgänge `OUT1`, `OUT2`, `OUT3`.
- **DummyGroup**: neutraler Platzhalter-Slot
  - **Typ**: `MyLib_AX_Option::sys::DummyGroup` (SubApp-Instanz)
  - **Parameter**: keine
  - **Erklärung**: Ohne reale I/O und ohne Wirkung; Zustand „nichts ausgewählt“.

- **OptionalGroup_001a**: erste Optionsgruppe mit Weiterreichung
  - **Typ**: `MyLib_AX_Option::sys::OptionalGroup_001a` (SubApp-Instanz)
  - **Parameter**: keine
  - **Erklärung**: Schaltet `Input_I1`/`Output_Q1` frei und reicht die Aktivierung über den Plug `AOption_OUT` weiter.
- **OptionalGroup_001b**: zweite Optionsgruppe (Ende der Kette)
  - **Typ**: `MyLib_AX_Option::sys::OptionalGroup_001b` (SubApp-Instanz)
  - **Parameter**: keine
  - **Erklärung**: Baugleich zu `OptionalGroup_001a`, aber ohne Plug; reicht nichts weiter.
- **OptionalGroup_002**: weitere Optionsgruppe
  - **Typ**: `MyLib_AX_Option::sys::OptionalGroup_002` (SubApp-Instanz)
  - **Parameter**: keine
  - **Erklärung**: Gruppe mit `I2`/`Q2`; wird über den Options-Adapter aktiviert bzw. deaktiviert.

Programmablauf und Verbindungen

1. `NumericValue_TO_AUI2_1.VALUEO` ⟶ `AOption_AUI_DEMUX_3.K`: Der Wert des Eingabefelds wählt den aktiven Ausgang.
2. `AOption_AUI_DEMUX_3.OUT1` ⟶ `DummyGroup.AOption_IN`, `OUT2` ⟶ `OptionalGroup_001a.AOption_IN`, `OUT3` ⟶ `OptionalGroup_002.AOption_IN`.
3. `OptionalGroup_001a.AOption_OUT` ⟶ `OptionalGroup_001b.AOption_IN`: Die Aktivierung wird an die zweite Gruppe weitergereicht.

Verhalten: 1 = nichts ausgewählt, 2 = `OptionalGroup_001a` und anschließend `OptionalGroup_001b`, 3 = `OptionalGroup_002`. Die Bestätigung an den DEMUX erfolgt erst, wenn die gesamte Kette (001a und 001b) initialisiert ist.

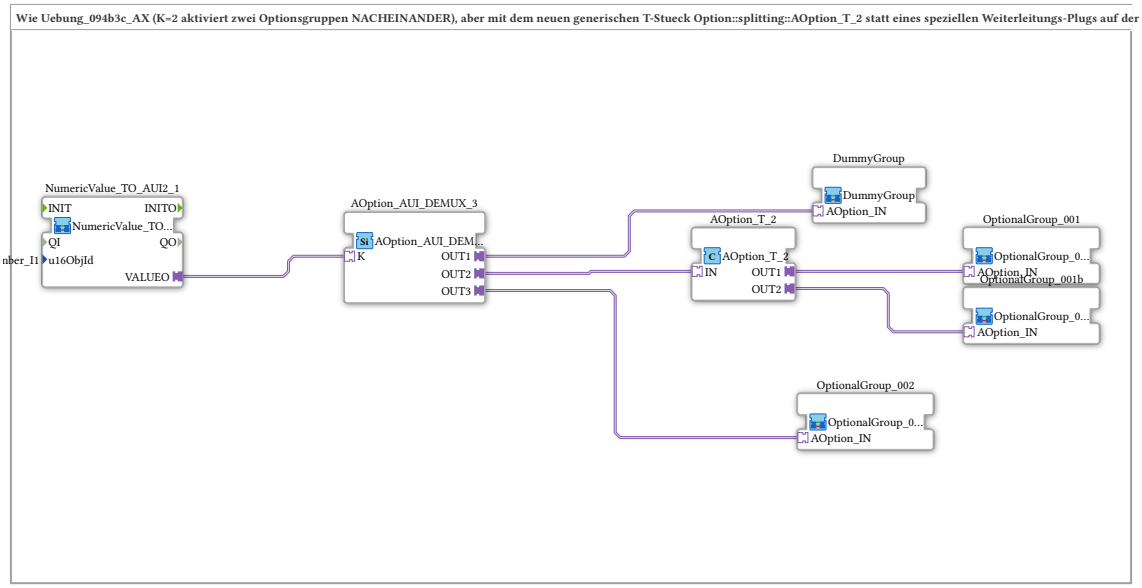
Zusammenfassung

Gegenüber `Uebung_094b3_AX` ist `OptionalGroup_001` durch `OptionalGroup_001a` ersetzt und mit der neuen `OptionalGroup_001b` verkettet. So werden mehrere Gruppen nacheinander aktiviert, bevor der Aufrufer seine Bestätigung erhält.

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

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

2.1.2.406 Uebung_094b3d_AX: Optionsauswahl mit zwei nacheinander aktivierten Optionsgruppen über das T-Stück AOption_T_2



• – ○ • – ○ • – ○ •

Einleitung

Die Übung entspricht Uebung_094b3c_AX: Der Index K = 2 aktiviert zwei Optionsgruppen NACHEINANDER. Statt eines speziellen Weiterleitungs-Plugs auf der Optionsgruppe selbst (AOption_OUT bei OptionalGroup_001a) wird hier das neue generische T-Stück AOption_T_2 verwendet. OptionalGroup_001 und OptionalGroup_001b sind beide die einfache, unveränderte Variante mit nur einem AOption_IN-Socket. Das T-Stück sitzt extern zwischen DEMUX und den beiden Gruppen und kann genauso an eine dritte Gruppe (oder ein weiteres T-Stück) weitergereicht werden. Die Aktivierung erfolgt sequenziell: zuerst OUT2 (OptionalGroup_001), danach erst OUT1 (OptionalGroup_001b), nicht gleichzeitig.

Verwendete Funktionsbausteine (FBs)

- **NumericValue_TO_AUI2_1**: liest das VT-Eingabefeld und stellt es als bidirektionalen AUI2-Plug bereit
  - **Typ**: MyLib::sys::NumericValue_TO_AUI2 (SubApp-Instanz)
  - **Parameter**: u16ObjId = InputNumber_I1
  - **Erklärung**: Liest das VT-Eingabefeld und reicht den Wert über VALUEO an den DEMUX weiter.
- **AOption_AUI_DEMUX_3**: Optionsauswahl mit drei Ausgängen
  - **Typ**: Option::selection::AOption_AUI_DEMUX_3
  - **Parameter**: keine
  - **Erklärung**: Aktiviert abhängig vom Index K genau einen der Ausgänge OUT1, OUT2, OUT3.
- **AOption_T_2**: Options-T-Stück
  - **Typ**: Option::splitting::AOption_T_2
  - **Parameter**: keine
  - **Erklärung**: Wird am IN-Socket vom DEMUX-Ausgang OUT2 gespeist und aktiviert sequenziell zuerst OUT2 (OptionalGroup_001), nach dessen INIT_CNF erst OUT1 (OptionalGroup_001b). OUT1 kann an den IN-Socket eines weiteren AOption_T_2 angeschlossen werden, um die Kette zu kaskadieren.
- **OptionalGroup_001**: erste Optionsgruppe
  - **Typ**: MyLib_AX_Option::sys::OptionalGroup_001 (SubApp-Instanz)
  - **Parameter**: keine
  - **Erklärung**: Schaltet Input_I1/Output_Q1 frei; nur ein AOption_IN-Socket.
- **OptionalGroup_001b**: zweite Optionsgruppe
  - **Typ**: MyLib_AX_Option::sys::OptionalGroup_001b (SubApp-Instanz)
  - **Parameter**: keine
  - **Erklärung**: Baugleich zu OptionalGroup_001 (ebenfalls Input_I1/Output_Q1); reicht nichts weiter.
- **OptionalGroup_002**: weitere Optionsgruppe
  - **Typ**: MyLib_AX_Option::sys::OptionalGroup_002 (SubApp-Instanz)
  - **Parameter**: keine
  - **Erklärung**: Gruppe mit I2/Q2; wird über den Options-Adapter aktiviert bzw. deaktiviert.
- **DummyGroup**: neutraler Platzhalter-Slot
  - **Typ**: MyLib_AX_Option::sys::DummyGroup (SubApp-Instanz)
  - **Parameter**: keine
  - **Erklärung**: Ohne reale I/O und ohne Wirkung; Zustand „nichts ausgewählt“.

Programmablauf und Verbindungen

1. NumericValue_TO_AUI2_1.VALUEO ⟶ AOption_AUI_DEMUX_3.K: Der Wert des Eingabefelds wählt den aktiven Ausgang.
2. AOption_AUI_DEMUX_3.OUT1 ⟶ DummyGroup.AOption_IN, AOption_AUI_DEMUX_3.OUT2 ⟶ AOption_T_2.IN, AOption_AUI_DEMUX_3.OUT3 ⟶ OptionalGroup_002.AOption_IN.
3. AOption_T_2.OUT2 ⟶ OptionalGroup_001.AOption_IN und AOption_T_2.OUT1 ⟶ OptionalGroup_001b.AOption_IN: Das T-Stück aktiviert zuerst OptionalGroup_001, danach OptionalGroup_001b.

Verhalten: K = 1 = nichts ausgewählt (DummyGroup), K = 2 = OptionalGroup_001 und anschließend OptionalGroup_001b (sequenziell über AOption_T_2), K = 3 = OptionalGroup_002.

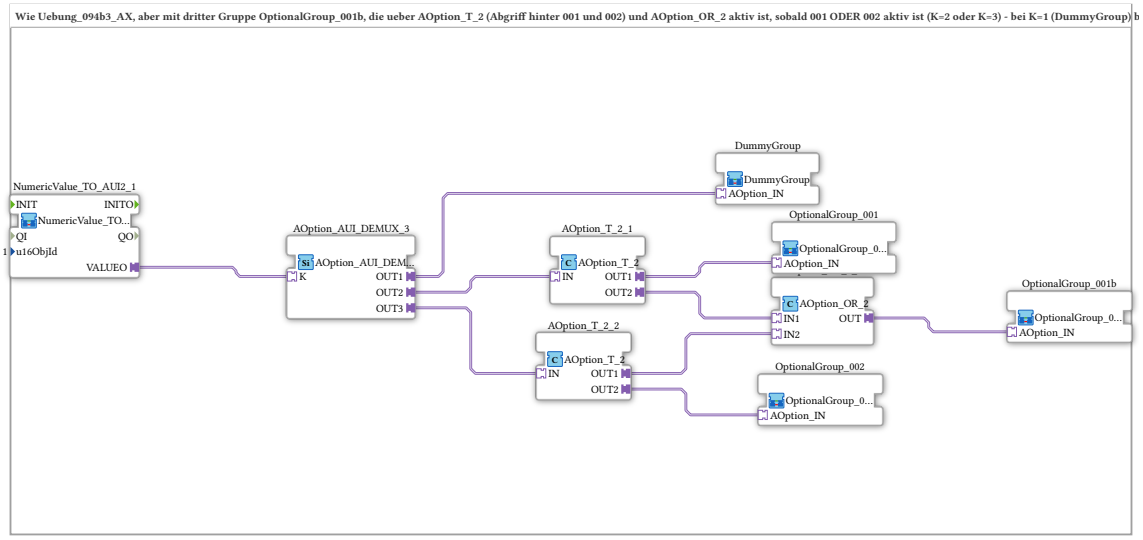
Zusammenfassung

Gegenüber Uebung_094b3c_AX entfällt die spezielle Gruppe OptionalGroup_001a mit AOption_OUT. Stattdessen verteilt das generische, kaskadierbare T-Stück AOption_T_2 die Aktivierung extern und sequenziell auf zwei einfache Optionsgruppen.

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

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

2.1.2.407 Uebung_094b3e_AX: Optionsauswahl mit dritter Gruppe hinter 001 und 002 (T-Stück und ODER)



• – ○ • – ○ • – ○ •

Einleitung

Die Übung baut auf `Uebung_094b3_AX` auf (`K = 1` Dummy, `K = 2` Gruppe 001, `K = 3` Gruppe 002). Zusätzlich wird eine dritte Gruppe `OptionalGroup_001b` aktiviert, sobald `OptionalGroup_001` ODER `OptionalGroup_002` aktiv ist, nicht aber bei `DummyGroup`. Da ein DEMUX-Ausgang nur an einen Socket angeschlossen werden kann, sitzt hinter `OUT2` und `OUT3` je ein `AOption_T_2`. Dessen zweiter Ausgang speist `AOption_OR_2`, dessen `OUT` die dritte Gruppe aktiviert.

Verwendete Funktionsbausteine (FBs)

- **NumericValue_TO_AUI2_1**: liest das VT-Eingabefeld und stellt es als bidirektionalen AUI2-Plug bereit
  - **Typ**: `MyLib::sys::NumericValue_TO_AUI2` (SubApp-Instanz)
  - **Parameter**: `u16ObjId = InputNumber_I1`
  - **Erklärung**: Liest das VT-Eingabefeld und reicht den Wert über `VALUEO` an den DEMUX weiter.
- **AOption_AUI_DEMUX_3**: Optionsauswahl mit drei Ausgängen
  - **Typ**: `Option::selection::AOption_AUI_DEMUX_3`
  - **Parameter**: keine
  - **Erklärung**: Aktiviert abhängig vom Index `K` genau einen der Ausgänge `OUT1`, `OUT2`, `OUT3`.
- **AOption_T_2_1 / AOption_T_2_2**
  - **Typ**: `Option::splitting::AOption_T_2`
  - **Parameter**: keine
  - **Erklärung**: Gespeist von `OUT2` bzw. `OUT3` des DEMUX; `OUT2` aktiviert die Gruppe 001 bzw. 002, danach `OUT1` den ODER-Eingang.
- **AOption_OR_2_1**
  - **Typ**: `Option::merging::AOption_OR_2`
  - **Parameter**: keine
  - **Erklärung**: `OUT` ist aktiv, wenn `IN1` oder `IN2` aktiv ist, und aktiviert die dritte Gruppe.
- **OptionalGroup_001**
  - **Typ**: `MyLib_AX_Option::sys::OptionalGroup_001` (SubApp-Instanz)
  - **Parameter**: keine
  - **Erklärung**: Schaltet `Input_I1`/`Output_Q1` frei.
- **OptionalGroup_002**
  - **Typ**: `MyLib_AX_Option::sys::OptionalGroup_002` (SubApp-Instanz)

  - **Parameter**: keine
  - **Erklärung**: Schaltet `Input_I2/Output_Q2` frei.
- **OptionalGroup_001b**
  - **Typ**: `MyLib_AX_Option::sys::OptionalGroup_001b` (SubApp-Instanz)
  - **Parameter**: keine
  - **Erklärung**: Dritte Gruppe (`Input_I1/Output_Q1`, baugleich zu 001); aktiv bei `K = 2` oder `K = 3`.
- **DummyGroup**
  - **Typ**: `MyLib_AX_Option::sys::DummyGroup` (SubApp-Instanz)
  - **Parameter**: keine
  - **Erklärung**: Ohne reale I/O; Zustand „nichts ausgewählt“.

Programmablauf und Verbindungen

1. `NumericValue_TO_AUI2_1.VALUEO` ⟶ `AOption_AUI_DEMUX_3.K`: Der Wert des Eingabefelds wählt den aktiven Ausgang.
2. `OUT1` ⟶ `DummyGroup.AOption_IN`, `OUT2` ⟶ `AOption_T_2_1.IN`, `OUT3` ⟶ `AOption_T_2_2.IN`.
3. `AOption_T_2_1.OUT2` ⟶ `OptionalGroup_001.AOption_IN`, `AOption_T_2_2.OUT2` ⟶ `OptionalGroup_002.AOption_IN`.
4. `AOption_T_2_1.OUT1` ⟶ `AOption_OR_2_1.IN1`, `AOption_T_2_2.OUT1` ⟶ `AOption_OR_2_1.IN2`.
5. `AOption_OR_2_1.OUT` ⟶ `OptionalGroup_001b.AOption_IN`.

Verhalten: `K = 1` = `DummyGroup` (die dritte Gruppe bleibt aus), `K = 2` = `OptionalGroup_001` und `OptionalGroup_001b`, `K = 3` = `OptionalGroup_002` und `OptionalGroup_001b`. Weil `AOption_OR_2` erst nach `OUT.INIT_CNF` an den anfragenden Socket bestätigt, meldet der DEMUX den Wechsel erst als fertig, wenn auch `OptionalGroup_001b` fertig ist.

Hinweis: `OptionalGroup_001b` benutzt wie `OptionalGroup_001` die I/O `Input_I1/Output_Q1`. Die Übung ist ein Verdrahtungstest und noch nicht auf dem Gerät geprüft.

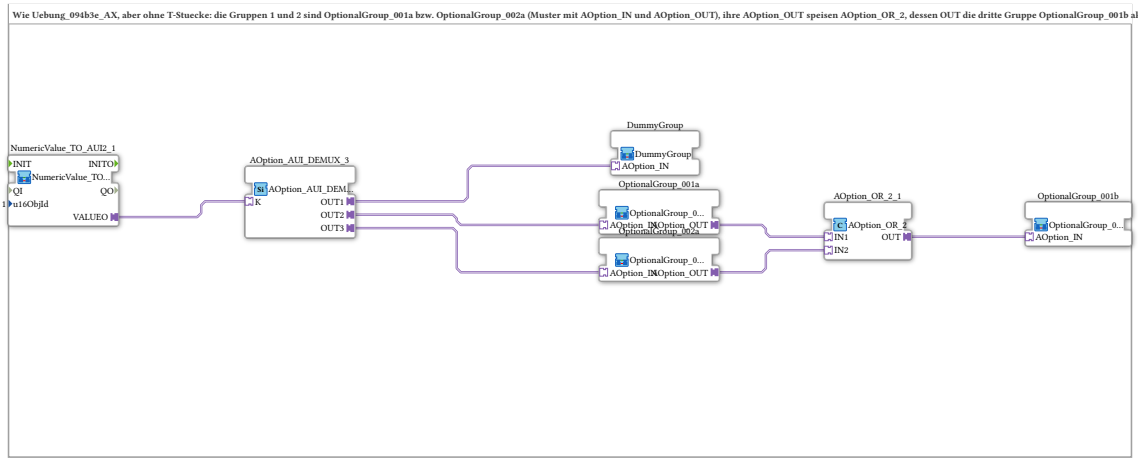
Zusammenfassung

Ein DEMUX-Ausgang speist nur einen Socket. Deshalb verteilen zwei `AOption_T_2` die Aktivierung auf die Gruppen und auf das ODER, und `AOption_OR_2` aktiviert die gemeinsame dritte Gruppe bei `K = 2` oder `K = 3`.

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

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

2.1.2.408 Uebung_094b3f_AX: Optionsauswahl mit dritter Gruppe über OptionalGroup_001a/002a und ODER



• – ○ • – ○ • – ○ •

Einleitung

Die Übung entspricht `Uebung_094b3e_AX`, kommt aber ohne T-Stücke aus: Die Gruppen 1 und 2 sind `OptionalGroup_001a` und `OptionalGroup_002a` mit `AOption_IN` und `AOption_OUT`. Deren `AOption_OUT`

gehen an AOption_OR_2, dessen OUT die dritte Gruppe OptionalGroup_001b aktiviert. DummyGroup ist nicht mit dem ODER verbunden.

Verwendete Funktionsbausteine (FBs)

- **NumericValue_TO_AUI2_1**: liest das VT-Eingabefeld und stellt es als bidirektionalen AUI2-Plug bereit
  - **Typ**: MyLib::sys::NumericValue_TO_AUI2 (SubApp-Instanz)
  - **Parameter**: u16ObjId = InputNumber_I1
  - **Erklärung**: Liest das VT-Eingabefeld und reicht den Wert über VALUEO an den DEMUX weiter.
- **AOption_AUI_DEMUX_3**: Optionsauswahl mit drei Ausgängen
  - **Typ**: Option::selection::AOption_AUI_DEMUX_3
  - **Parameter**: keine
  - **Erklärung**: Aktiviert abhängig vom Index K genau einen der Ausgänge OUT1, OUT2, OUT3.
- **OptionalGroup_001a**
  - **Typ**: MyLib_AX_Option::sys::OptionalGroup_001a (SubApp-Instanz)
  - **Parameter**: keine
  - **Erklärung**: Gruppe 1 (Input_I1/Output_Q1) mit AOption_IN und AOption_OUT.
- **OptionalGroup_002a**
  - **Typ**: MyLib_AX_Option::sys::OptionalGroup_002a (SubApp-Instanz)
  - **Parameter**: keine
  - **Erklärung**: Gruppe 2 (Input_I2/Output_Q2) mit AOption_IN und AOption_OUT.
- **AOption_OR_2_1**
  - **Typ**: Option::merging::AOption_OR_2
  - **Parameter**: keine
  - **Erklärung**: OUT ist aktiv, wenn IN1 oder IN2 aktiv ist.
- **OptionalGroup_001b**
  - **Typ**: MyLib_AX_Option::sys::OptionalGroup_001b (SubApp-Instanz)
  - **Parameter**: keine
  - **Erklärung**: Dritte Gruppe (Input_I1/Output_Q1), Endglied.
- **DummyGroup**
  - **Typ**: MyLib_AX_Option::sys::DummyGroup (SubApp-Instanz)
  - **Parameter**: keine
  - **Erklärung**: Ohne reale I/O; Zustand „nichts ausgewählt“.

Programmablauf und Verbindungen

1. NumericValue_TO_AUI2_1.VALUEO ⟶ AOption_AUI_DEMUX_3.K.
2. OUT1 ⟶ DummyGroup.AOption_IN, OUT2 ⟶ OptionalGroup_001a.AOption_IN, OUT3 ⟶ OptionalGroup_002a.AOption_IN.
3. OptionalGroup_001a.AOption_OUT ⟶ AOption_OR_2_1.IN1, OptionalGroup_002a.AOption_OUT ⟶ AOption_OR_2_1.IN2.
4. AOption_OR_2_1.OUT ⟶ OptionalGroup_001b.AOption_IN.

Verhalten: K = 1 = DummyGroup (dritte Gruppe aus), K = 2 = OptionalGroup_001a und danach OptionalGroup_001b, K = 3 = OptionalGroup_002a und danach OptionalGroup_001b. Die a-Gruppen melden INIT_CNF erst zurück, wenn die nachgeschaltete Kette bestätigt hat.

Hinweis: OptionalGroup_001b benutzt dieselben I/O wie OptionalGroup_001a. Die Übung ist ein Verdrahtungstest und noch nicht auf dem Gerät geprüft.

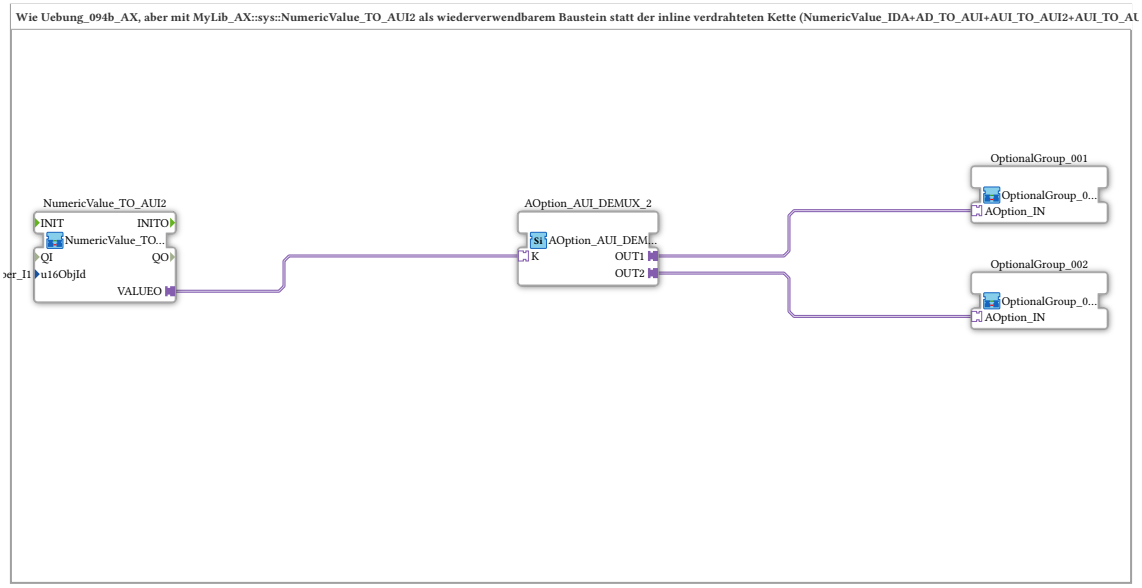
Zusammenfassung

Die Gruppen reichen die Aktivierung selbst über AOption_OUT weiter, sodass keine T-Stücke nötig sind. AOption_OR_2 aktiviert die dritte Gruppe, sobald eine der beiden aktiv ist.

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

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

2.1.2.409 Uebung_094b4_AX: Optionsauswahl (2 Optionen) mit wiederverwendbarem Baustein NumericValue_TO_AUI2



• – ◦ • – ◦ • – ◦ •

Einleitung

Die Übung entspricht `Uebung_094b_AX`, verwendet aber den wiederverwendbaren Baustein `MyLib::sys::NumericValue_TO_AUI2` statt der inline verdrahteten Kette aus `NumericValue_IDA`, `AD_TO_AUI`, `AUI_TO_AUI2`, `AUI_TO_AUDI` und `Q_NumericValue_AUDI`. Die Objekt-ID des VT-Felds wird als SubApp-Parameter übergeben. Ausgewählt wird zwischen zwei Optionsgruppen über `AOption_AUI_DEMUX_2` (Index, Einzelauswahl).

Verwendete Funktionsbausteine (FBs)

- **NumericValue_TO_AUI2**: liest das VT-Eingabefeld und stellt es als bidirektionalen AUI2-Plug bereit
  - **Typ**: `MyLib::sys::NumericValue_TO_AUI2` (SubApp-Instanz)
  - **Parameter**: `u16ObjId = InputNumber_I1`
  - **Erklärung**: Liest das VT-Feld und reicht den Wert über `VALUEO` weiter. Der Consumer meldet den übernommenen Wert über denselben Plug zurück, das Echo wird ins VT-Feld zurückgeschrieben.
- **AOption_AUI_DEMUX_2**: Optionsauswahl mit zwei Ausgängen
  - **Typ**: `Option::selection::AOption_AUI_DEMUX_2`
  - **Parameter**: keine
  - **Erklärung**: Aktiviert abhängig vom Index `K` genau einen der Ausgänge `OUT1` oder `OUT2`.
- **OptionalGroup_001**, **OptionalGroup_002**: Optionsgruppen
  - **Typ**: `MyLib_AX_Option::sys::OptionalGroup_001` bzw. `MyLib_AX_Option::sys::OptionalGroup_002` (SubApp-Instanzen)
  - **Parameter**: keine
  - **Erklärung**: Gruppe 1 enthält `I1`/`Q1`, Gruppe 2 enthält `I2`/`Q2`.

Programmablauf und Verbindungen

1. `NumericValue_TO_AUI2.VALUEO` ⟶ `AOption_AUI_DEMUX_2.K`: Der Wert des Eingabefelds wählt den aktiven Ausgang.
2. `AOption_AUI_DEMUX_2.OUT1` ⟶ `OptionalGroup_001.AOption_IN`, `OUT2` ⟶ `OptionalGroup_002.AOption_IN`.

Verhalten: Der Index 1 aktiviert Gruppe 1, der Index 2 aktiviert Gruppe 2; es ist immer nur eine Gruppe aktiv.

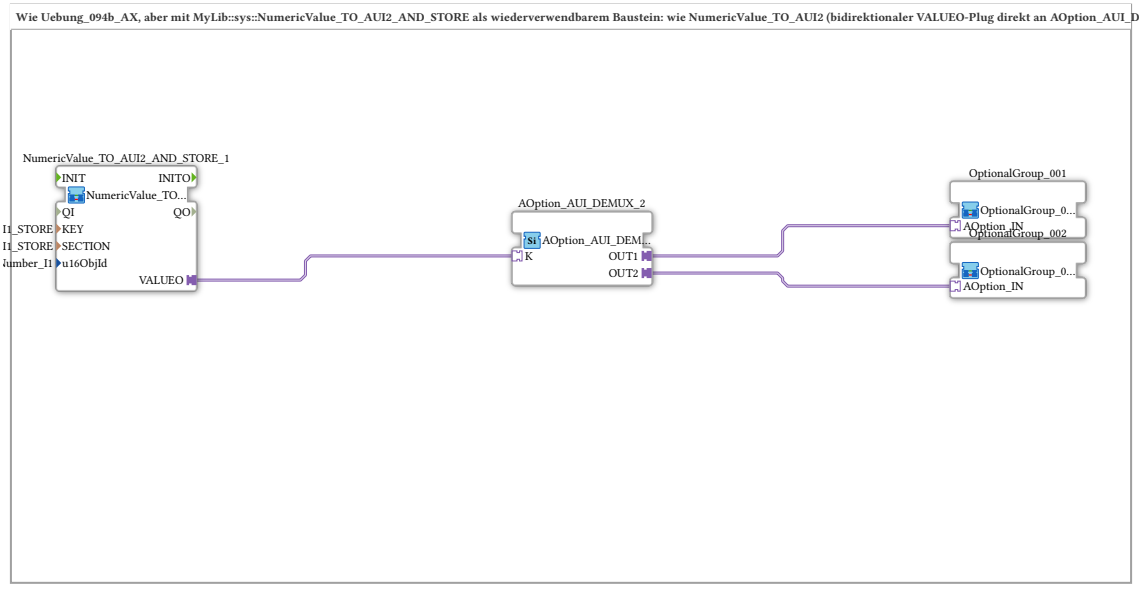
Zusammenfassung

Die Lese-/Echo-Kette ist in `MyLib::sys::NumericValue_TO_AUI2` gekapselt. Die Übung verhält sich wie `Uebung_094b_AX` bei deutlich kleinerem Netzwerk. Die Variante mit INI-Persistenz ist `Uebung_094b4b_AX`.

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

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

2.1.2.410 Uebung_094b4b_AX: Optionsauswahl (2 Optionen) mit INI-persistiertem Index



• – ◦ • – ◦ • – ◦ •

Einleitung

Die Übung entspricht `Uebung_094b4_AX`, verwendet aber `MyLib::sys::NumericValue_TO_AUI2_AND_STORE` statt `NumericValue_TO_AUI2`: Der bidirektionale `VALUEO`-Plug geht direkt an `AOption_AUI_DEMUX_2.K`, zusätzlich wird der Index per INI persistiert. Die Konstanten `KEY_I1_STORE` und `SECTION_I1_STORE` sind bereits aus `Uebung_012c_AUDI` bekannt.

Verwendete Funktionsbausteine (FBs)

- **NumericValue_TO_AUI2_AND_STORE_1**: VT-Eingabefeld lesen, als AUI2-Plug bereitstellen und in INI speichern
  - **Typ**: `MyLib::sys::NumericValue_TO_AUI2_AND_STORE` (SubApp-Instanz)
  - **Parameter**: `KEY = KEY_I1_STORE`, `SECTION = SECTION_I1_STORE`, `u16ObjId = InputNumber_I1`
  - **Erklärung**: Wie `NumericValue_TO_AUI2`, zusätzlich mit INI-Persistenz: Beim Boot gewinnt der aus der INI geladene Wert gegen die Live-VT-Lesung, danach jede Live-Änderung.
- **AOption_AUI_DEMUX_2**: Optionsauswahl mit zwei Ausgängen
  - **Typ**: `Option::selection::AOption_AUI_DEMUX_2`
  - **Parameter**: keine
  - **Erklärung**: Aktiviert abhängig vom Index `K` genau einen der Ausgänge `OUT1` oder `OUT2`.
- **OptionalGroup_001**, **OptionalGroup_002**: Optionsgruppen
  - **Typ**: `MyLib_AX_Option::sys::OptionalGroup_001` bzw. `MyLib_AX_Option::sys::OptionalGroup_002` (SubApp-Instanzen)
  - **Parameter**: keine
  - **Erklärung**: Gruppe 1 enthält `I1/Q1`, Gruppe 2 enthält `I2/Q2`.

Programmablauf und Verbindungen

1. `NumericValue_TO_AUI2_AND_STORE_1.VALUEO` → `AOption_AUI_DEMUX_2.K`: Der (ggf. aus der INI geladene) Index wählt den aktiven Ausgang.
2. `AOption_AUI_DEMUX_2.OUT1` → `OptionalGroup_001.AOption_IN`, `OUT2` → `OptionalGroup_002.AOption_IN`.

Verhalten: Wie `Uebung_094b4_AX` (Index 1 = Gruppe 1, Index 2 = Gruppe 2), die Auswahl bleibt zusätzlich über einen Neustart erhalten.

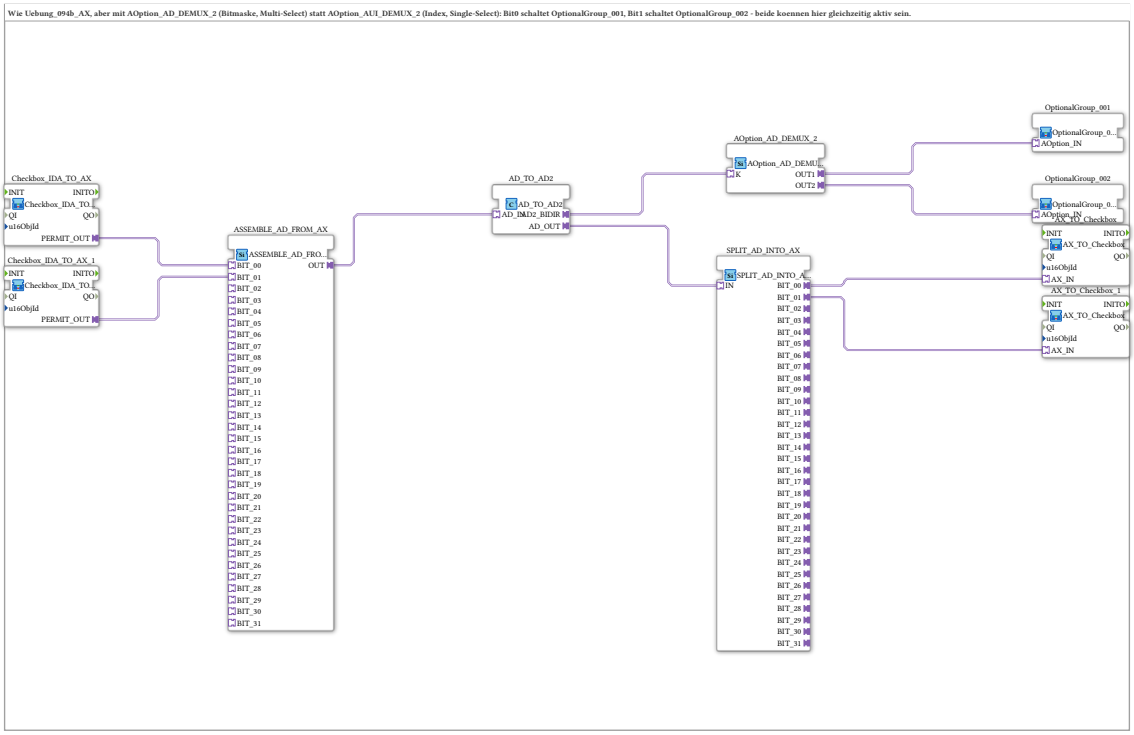
Zusammenfassung

Der Austausch durch die `_AND_STORE`-Variante macht die Optionsauswahl neustartfest, ohne das restliche Netzwerk zu ändern.

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

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

2.1.2.411 Uebung_094b5_AX: Optionsauswahl per Bitmaske (Mehrfachauswahl) mit AOption_AD_DEMUX_2



• – ○ • – ○ • – ○ •

Einleitung

Die Übung entspricht `Uebung_094b_AX`, verwendet aber `AOption_AD_DEMUX_2` (Bitmaske, Mehrfachauswahl) statt `AOption_AUI_DEMUX_2` (Index, Einzelauswahl): Bit 0 schaltet `OptionalGroup_001`, Bit 1 schaltet `OptionalGroup_002` beide können gleichzeitig aktiv sein. Die Bitmaske wird hier aus zwei VT-Checkboxen zusammengesetzt und der Zustand wieder in die Checkboxen zurückgeschrieben. `AD_TO_AUI` und `AUI_TO_AUI2` sind durch `AD_TO_AD2` ersetzt, da der Wert bereits als DWORD (AD) vorliegt.

Verwendete Funktionsbausteine (FBs)

- **Checkbox_IDA_TO_AX**, **Checkbox_IDA_TO_AX_1**: VT-Checkbox lesen
  - **Typ**: `MyLib::sys::Checkbox_IDA_TO_AX` (SubApp-Instanzen)
  - **Parameter**: keine gesetzt (`u16ObjId` bleibt auf dem Standardwert `ID_NULL`)
  - **Erklärung**: Reicht den Ein/Aus-Zustand einer VT-Checkbox als AX-Pegel über `PERMIT_OUT` weiter. Die Objekt-ID (`u16ObjId`, ObjectID der von der Checkbox referenzierten NumberVariable) muss für den realen Betrieb noch gesetzt werden.
- **ASSEMBLE_AD_FROM_AX**: Bitmaske zusammensetzen
  - **Typ**: `adapter::assembling::ASSEMBLE_AD_FROM_AX`
  - **Parameter**: keine
  - **Erklärung**: Setzt aus den AX-Eingängen `BIT_00` und `BIT_01` einen DWORD-Wert (AD) zusammen.
- **AD_TO_AD2**: Brücke zum bidirektionalen AD2-Adapter
  - **Typ**: `adapter::conversion::bidirectional::AD_TO_AD2`
  - **Parameter**: keine

  - **Erklärung**: Reicht die Bitmaske über `AD2_BIDIR` an den Consumer (`K` des DEMUX) weiter und liefert den vom Consumer übernommenen Wert über `AD_OUT` zurück.
- **AOption_AD_DEMUX_2**: Optionsauswahl per Bitmaske
  - **Typ**: `Option::selection::AOption_AD_DEMUX_2`
  - **Parameter**: keine
  - **Erklärung**: Bit 0 von `K` aktiviert `OUT1`, Bit 1 aktiviert `OUT2`; mehrere Ausgänge können gleichzeitig aktiv sein.
- **OptionalGroup_001**, **OptionalGroup_002**: Optionsgruppen
  - **Typ**: `MyLib_AX_Option::sys::OptionalGroup_001` bzw. `MyLib_AX_Option::sys::OptionalGroup_002` (SubApp-Instanzen)
  - **Parameter**: keine
  - **Erklärung**: Gruppe 1 enthält `I1`/`Q1`, Gruppe 2 enthält `I2`/`Q2`.
- **SPLIT_AD_INTO_AX**: Bitmaske zerlegen
  - **Typ**: `adapter::splitting::SPLIT_AD_INTO_AX`
  - **Parameter**: keine
  - **Erklärung**: Zerlegt den zurückgemeldeten DWORD-Wert in die AX-Pegel `BIT_00` und `BIT_01`.
- **AX_TO_Checkbox**, **AX_TO_Checkbox_1**: VT-Checkbox schreiben
  - **Typ**: `MyLib::sys::AX_TO_Checkbox` (SubApp-Instanzen)
  - **Parameter**: keine gesetzt (`u16ObjId` bleibt auf dem Standardwert `ID_NULL`)
  - **Erklärung**: Gegenstück zu `Checkbox_IDA_TO_AX`: schreibt den AX-Pegel von `AX_IN` bei jeder Änderung auf die VT-Checkbox.

Programmablauf und Verbindungen

1. `Checkbox_IDA_TO_AX.PERMIT_OUT` → `ASSEMBLE_AD_FROM_AX.BIT_00` und `Checkbox_IDA_TO_AX_1.PERMIT_OUT` → `ASSEMBLE_AD_FROM_AX.BIT_01`: Die beiden Checkboxen bilden Bit 0 und Bit 1 der Maske.
2. `ASSEMBLE_AD_FROM_AX.OUT` → `AD_TO_AD2.AD_IN`, `AD_TO_AD2.AD2_BIDIR` → `AOption_AD_DEMUX_2.K`: Die Maske wählt die aktiven Ausgänge.
3. `AOption_AD_DEMUX_2.OUT1` → `OptionalGroup_001.AOption_IN`, `OUT2` → `OptionalGroup_002.AOption_IN`.
4. Echo-Pfad: `AD_TO_AD2.AD_OUT` → `SPLIT_AD_INTO_AX.IN`; `BIT_00` → `AX_TO_Checkbox.AX_IN`, `BIT_01` → `AX_TO_Checkbox_1.AX_IN`: Der übernommene Wert wird in die Checkboxen zurückgeschrieben.

Verhalten: Checkbox 1 schaltet Gruppe 1, Checkbox 2 schaltet Gruppe 2; beide Gruppen können gleichzeitig aktiv sein.

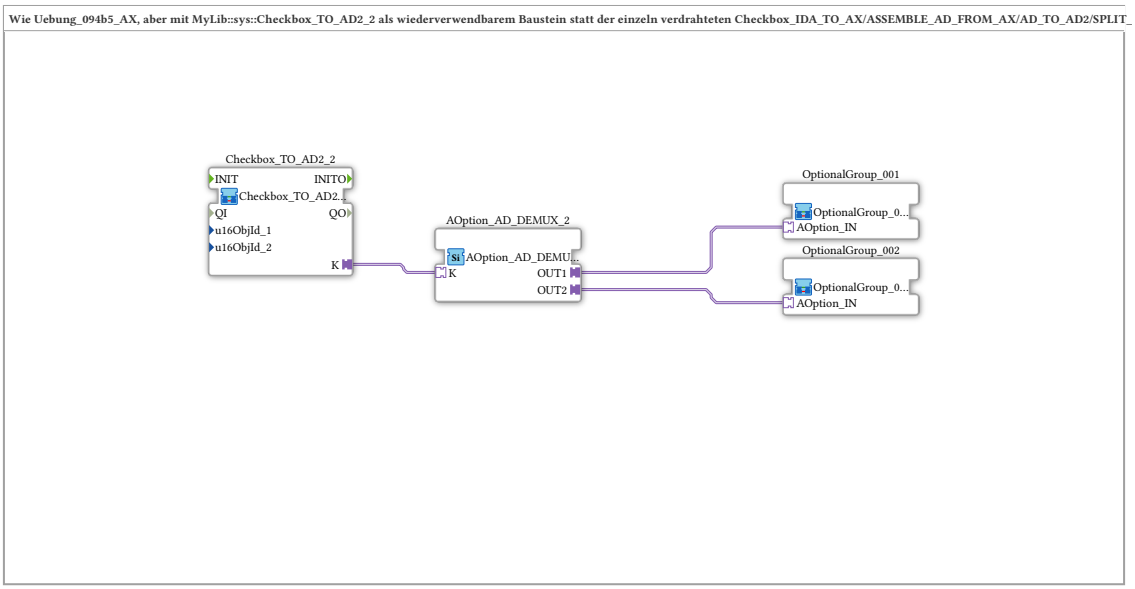
Zusammenfassung

Mit `AOption_AD_DEMUX_2` wird aus der Einzelauswahl der Vorgängerübungen eine Mehrfachauswahl per Bitmaske. `ASSEMBLE_AD_FROM_AX` und `SPLIT_AD_INTO_AX` verbinden die einzelnen Checkbox-Pegel mit der Bitmaske und zurück.

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

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

2.1.2.412 Uebung_094b5b_AX: Checkbox-Kette als wiederverwendbarer Baustein



• – ◦ • – ◦ • – ◦ •

Einleitung

Zweite Version von `Uebung_094b5_AX`: Alles außer `AOption_AD_DEMUX_2`, `OptionalGroup_001` und `OptionalGroup_002` wurde in den wiederverwendbaren Baustein `MyLib::sys::Checkbox_TO_AD2_2` ausgelagert. Das Netzwerk besteht dadurch nur noch aus vier Instanzen.

Verwendete Funktionsbausteine (FBs)

- **Checkbox_TO_AD2_2**: Wiederverwendbarer Baustein für zwei VT-Checkboxen
  - **Typ**: `MyLib::sys::Checkbox_TO_AD2_2` (SubApp)
  - **Parameter**: keine
  - **Erklärung**: Fasst zwei VT-Checkboxen zu einer Bitmaske zusammen und stellt sie am bidirektionalen Plug `K` bereit. Ersetzt die einzeln verdrahtete Kette `Checkbox_IDA_TO_AX`/`ASSEMBLE_AD_FROM_AX`/`AD_TO_AD2`/`SPLIT_AD_INTO_AX`/`AX_TO_Checkbox` aus `Uebung_094b5_AX`.
- **AOption_AD_DEMUX_2**: Auswahl von Optionsgruppen
  - **Typ**: `Option::selection::AOption_AD_DEMUX_2`
  - **Parameter**: keine
  - **Erklärung**: Empfängt die Bitmaske am Socket `K` und gibt über `OUT1`/`OUT2` die Optionsgruppen frei.
- **OptionalGroup_001**: Optionsgruppe 1
  - **Typ**: `MyLib_AX_Option::sys::OptionalGroup_001` (SubApp)
  - **Parameter**: keine
  - **Erklärung**: Wird über den Adapter `AOption_IN` freigegeben (QI anstelle Permit).
- **OptionalGroup_002**: Optionsgruppe 2
  - **Typ**: `MyLib_AX_Option::sys::OptionalGroup_002` (SubApp)
  - **Parameter**: keine
  - **Erklärung**: Wird über den Adapter `AOption_IN` freigegeben (QI anstelle Permit).

Programmablauf und Verbindungen

1. `Checkbox_TO_AD2_2.K` ⟶ `AOption_AD_DEMUX_2.K` (AdapterConnection): die Bitmaske der beiden Checkboxen gelangt zum Demultiplexer.
2. `AOption_AD_DEMUX_2.OUT1` ⟶ `OptionalGroup_001.AOption_IN`: Freigabe der Optionsgruppe 1.
3. `AOption_AD_DEMUX_2.OUT2` ⟶ `OptionalGroup_002.AOption_IN`: Freigabe der Optionsgruppe 2.

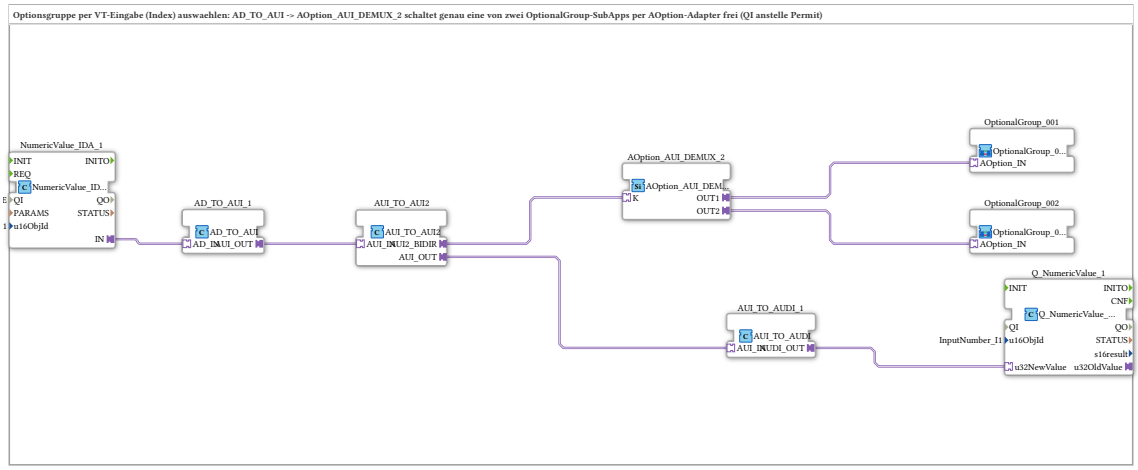
Zusammenfassung

Funktional entspricht die Übung `Uebung_094b5_AX`, ist aber durch die Auslagerung in `Checkbox_TO_AD2_2` deutlich übersichtlicher.

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

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

2.1.2.413 Uebung_094b_AX: Optionsgruppe per VT-Eingabe (Index) auswählen



• – ○ • – ○ • – ○ •

Einleitung

Ein VT-Eingabefeld (`InputNumber_I1`) wählt per Index eine von zwei Optionsgruppen aus: `AOption_AUI_DEMUX_2` schaltet genau eine der beiden `OptionalGroup`-SubApps per `AOption`-Adapter frei (`QI` anstelle von Permit). Der vom Consumer zurückgemeldete Wert wird zusätzlich in das Eingabefeld zurückgeschrieben (Echo).

Verwendete Funktionsbausteine (FBs)

- **NumericValue_IDA_1**: VT-Eingabefeld lesen
  - **Typ**: `isobus::UT::io::NumericValue::NumericValue_IDA`
  - **Parameter**: `QI = TRUE`, `u16ObjId = InputNumber_I1`
  - **Erklärung**: Liest den Wert des VT-Eingabefelds `InputNumber_I1` und gibt ihn über den Adapter `IN` aus.
- **AD_TO_AUI_1**: Adapterwandler AD nach AUI
  - **Typ**: `adapter::conversion::unidirectional::AD_TO_AUI`
  - **Parameter**: keine
  - **Erklärung**: Wandelt den Adapter `AD_IN` des Eingabefelds in einen `AUI_OUT`.
- **AUI_TO_AUI2**: Bidirektionale Brücke
  - **Typ**: `adapter::conversion::bidirectional::AUI_TO_AUI2`
  - **Parameter**: keine
  - **Erklärung**: Stellt den Index als bidirektionalen Stecker `AUI2_BIDIR` für den Consumer bereit und liefert über `AUI_OUT` den vom Consumer zurückgemeldeten Wert.
- **AOption_AUI_DEMUX_2**: Auswahl genau einer von zwei Optionsgruppen
  - **Typ**: `Option::selection::AOption_AUI_DEMUX_2`
  - **Parameter**: keine
  - **Erklärung**: Empfängt den Index am Socket `K` und gibt über `OUT1`/`OUT2` genau eine der beiden Optionsgruppen frei.
- **OptionalGroup_001**: Optionsgruppe 1
  - **Typ**: `MyLib_AX_Option::sys::OptionalGroup_001` (SubApp)
  - **Parameter**: keine
  - **Erklärung**: Wird über den Adapter `AOption_IN` freigegeben (QI anstelle Permit).
- **OptionalGroup_002**: Optionsgruppe 2
  - **Typ**: `MyLib_AX_Option::sys::OptionalGroup_002` (SubApp)
  - **Parameter**: keine
  - **Erklärung**: Wird über den Adapter `AOption_IN` freigegeben (QI anstelle Permit).
- **AUI_TO_AUDI_1**: Adapterwandler AUI nach AUDI
  - **Typ**: `adapter::conversion::unidirectional::AUI_TO_AUDI`

  - **Parameter**: keine
  - **Erklärung**: Wandelt den zurückgemeldeten Index in einen `AUDI_OUT` für die VT-Rückschreibung.
- **Q_NumericValue_1**: VT-Wert schreiben
  - **Typ**: `isobus::UT::Q::Q_NumericValue_AUDI`
  - **Parameter**: `u16ObjId = InputNumber_I1`
  - **Erklärung**: Schreibt den Wert über `u32NewValue` zurück in das Eingabefeld `InputNumber_I1`.

Programmablauf und Verbindungen

1. `NumericValue_IDA_1.IN` ⟶ `AD_TO_AUI_1.AD_IN` (AdapterConnection): der Wert des Eingabefelds wird in einen AUI-Strom gewandelt.
2. `AD_TO_AUI_1.AUI_OUT` ⟶ `AUI_TO_AUI2.AUI_IN`: der Wert wird in die bidirektionale Brücke eingespeist.
3. `AUI_TO_AUI2.AUI2_BIDIR` ⟶ `AOption_AUI_DEMUX_2.K`: der Index gelangt zum Demultiplexer.
4. `AOption_AUI_DEMUX_2.OUT1` ⟶ `OptionalGroup_001.AOption_IN` und `AOption_AUI_DEMUX_2.OUT2` ⟶ `OptionalGroup_002.AOption_IN`: genau eine Gruppe wird freigegeben.
5. `AUI_TO_AUI2.AUI_OUT` ⟶ `AUI_TO_AUDI_1.AUI_IN` ⟶ `Q_NumericValue_1.u32NewValue`: der zurückgemeldete Wert wird ins Eingabefeld geschrieben.

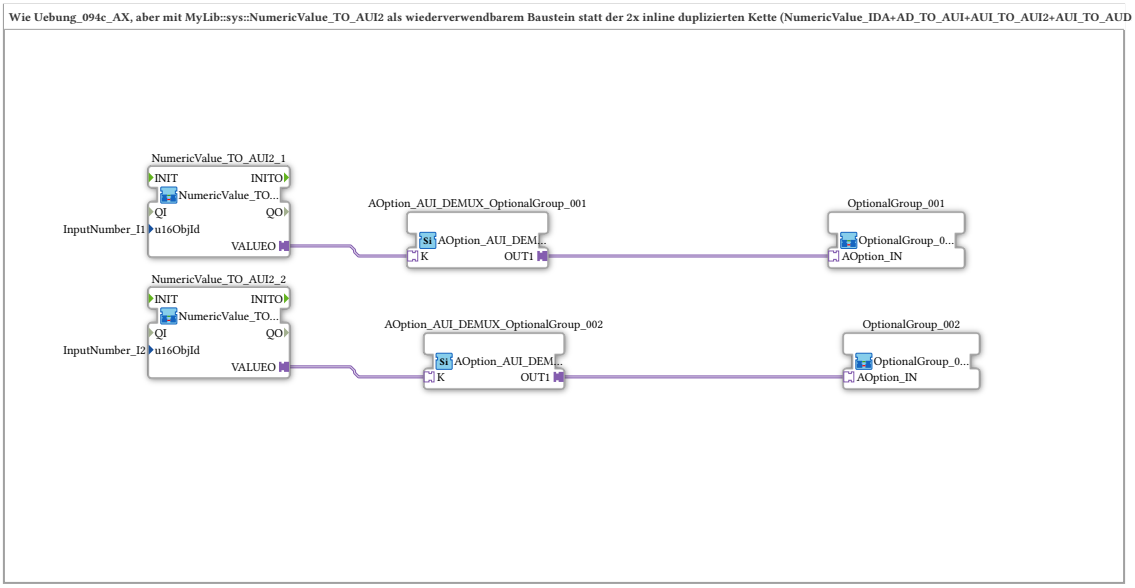
Zusammenfassung

Die Übung zeigt die Auswahl einer Optionsgruppe per VT-Eingabe: Der Index läuft über `AUI_TO_AUI2` in `AOption_AUI_DEMUX_2`, das genau eine von zwei Optionsgruppen per `QI` freigibt; die Rückmeldung wird ins Eingabefeld gespiegelt.

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

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

2.1.2.414 Uebung_094c2_AX: Dropdown-Kette als wiederverwendbarer Baustein



• – ○ • – ○ • – ○ •

Einleitung

Wie `Uebung_094c_AX`, aber die pro Dropdown duplizierte Lese-/Echo-Kette (`NumericValue_IDA`, `AD_TO_AUI`, `AUI_TO_AUI2`, `AUI_TO_AUDI`, `Q_NumericValue_AUDI`) wurde in den wiederverwendbaren Baustein `MyLib::sys::NumericValue_TO_AUI2` ausgelagert; `u16ObjId` ist dabei ein SubApp-Parameter statt fest verdrahtet. Weiterhin gibt es keinen gegenseitigen Ausschluss der beiden Gruppen.

Verwendete Funktionsbausteine (FBs)

- **NumericValue_TO_AUI2_1**: Wiederverwendbarer Baustein je Dropdown
  - **Typ**: `MyLib::sys::NumericValue_TO_AUI2` (SubApp)

  - **Parameter**: `u16ObjId = InputNumber_I1`
  - **Erklärung**: Liest das VT-Eingabefeld `InputNumber_I1` und stellt den Index als bidirektionalen Plug `VALUEO` bereit; die Rückschreibung ins VT ist enthalten.
- **NumericValue_TO_AUI2_2**: Wiederverwendbarer Baustein je Dropdown
  - **Typ**: `MyLib::sys::NumericValue_TO_AUI2` (SubApp)
  - **Parameter**: `u16ObjId = InputNumber_I2`
  - **Erklärung**: Liest das VT-Eingabefeld `InputNumber_I2` und stellt den Index als bidirektionalen Plug `VALUEO` bereit; die Rückschreibung ins VT ist enthalten.
- **AOption_AUI_DEMUX_OptionalGroup_001**: Freigabe für Optionsgruppe 1
  - **Typ**: `Option::selection::AOption_AUI_DEMUX_1`
  - **Parameter**: keine
  - **Erklärung**: Gibt über `OUT1` die Optionsgruppe 1 frei (Socket `K`).
- **AOption_AUI_DEMUX_OptionalGroup_002**: Freigabe für Optionsgruppe 2
  - **Typ**: `Option::selection::AOption_AUI_DEMUX_1`
  - **Parameter**: keine
  - **Erklärung**: Gibt über `OUT1` die Optionsgruppe 2 frei (Socket `K`).
- **OptionalGroup_001**: Optionsgruppe 1
  - **Typ**: `MyLib_AX_Option::sys::OptionalGroup_001` (SubApp)
  - **Parameter**: keine
  - **Erklärung**: Wird über den Adapter `AOption_IN` freigegeben (QI anstelle Permit).
- **OptionalGroup_002**: Optionsgruppe 2
  - **Typ**: `MyLib_AX_Option::sys::OptionalGroup_002` (SubApp)
  - **Parameter**: keine
  - **Erklärung**: Wird über den Adapter `AOption_IN` freigegeben (QI anstelle Permit).

Programmablauf und Verbindungen

1. `NumericValue_TO_AUI2_1.VALUEO` ⟶ `AOption_AUI_DEMUX_OptionalGroup_001.K`, danach `AOption_AUI_DEMUX_OptionalGroup_001.OUT1` ⟶ `OptionalGroup_001.AOption_IN` (Dropdown 1).
2. `NumericValue_TO_AUI2_2.VALUEO` ⟶ `AOption_AUI_DEMUX_OptionalGroup_002.K`, danach `AOption_AUI_DEMUX_OptionalGroup_002.OUT1` ⟶ `OptionalGroup_002.AOption_IN` (Dropdown 2).

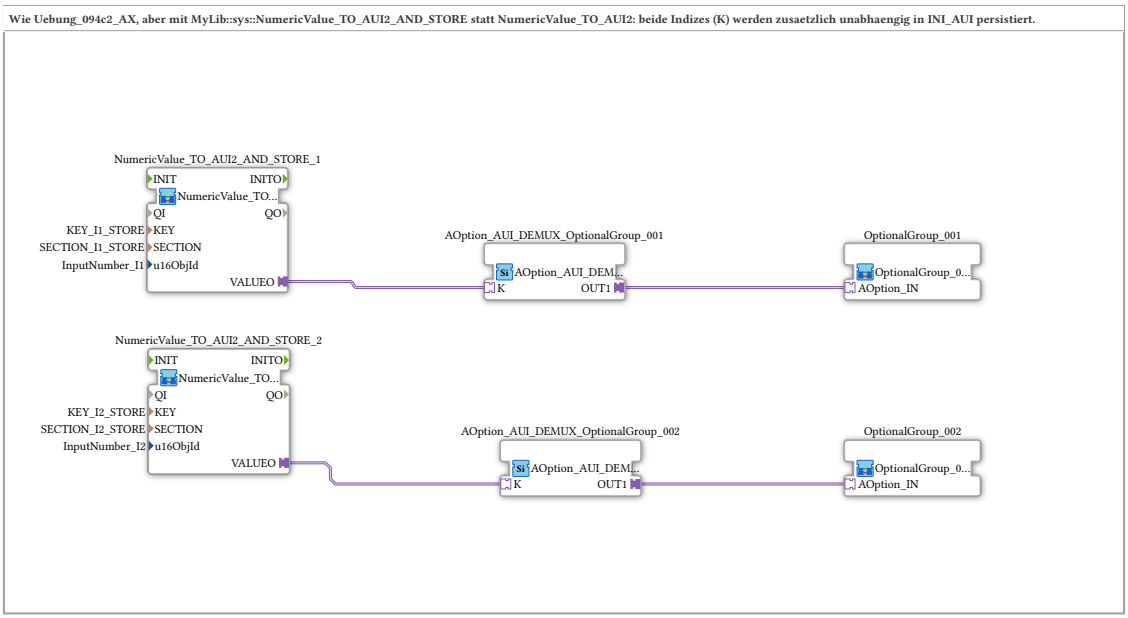
Zusammenfassung

Die Übung ist funktional gleich zu `Uebung_094c_AX`, aber mit nur sechs statt vierzehn Instanzen deutlich kompakter.

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

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

2.1.2.415 Uebung_094c2b_AX: Dropdown-Kette mit INI-Persistenz



• – ○ • – ○ • – ○ •

Einleitung

Wie `Uebung_094c2_AX`, aber mit `MyLib::sys::NumericValue_TO_AUI2_AND_STORE` statt `NumericValue_TO_AUI2`: gleicher bidirektionaler `VALUEO`-Plug, beide Indizes (K) werden zusätzlich unabhängig in `INI_AUI` persistiert (`KEY_I1_STORE`/`SECTION_I1_STORE` bzw. `KEY_I2_STORE`/`SECTION_I2_STORE`), analog `Uebung_094b3b_AX`/`Uebung_094b4b_AX`.

Verwendete Funktionsbausteine (FBs)

- **NumericValue_TO_AUI2_AND_STORE_1**: Wiederverwendbarer Baustein je Dropdown
  - **Typ**: `MyLib::sys::NumericValue_TO_AUI2_AND_STORE` (SubApp)
  - **Parameter**: `KEY = KEY_I1_STORE`, `SECTION = SECTION_I1_STORE`, `u16ObjId = InputNumber_I1`
  - **Erklärung**: Liest das VT-Eingabefeld `InputNumber_I1` und stellt den Index als bidirektionalen Plug `VALUEO` bereit; die Rückschreibung ins VT ist enthalten. Zusätzlich wird der Index unabhängig in `INI_AUI` persistiert (Schlüssel `KEY_I1_STORE`, Sektion `SECTION_I1_STORE`).
- **NumericValue_TO_AUI2_AND_STORE_2**: Wiederverwendbarer Baustein je Dropdown
  - **Typ**: `MyLib::sys::NumericValue_TO_AUI2_AND_STORE` (SubApp)
  - **Parameter**: `KEY = KEY_I2_STORE`, `SECTION = SECTION_I2_STORE`, `u16ObjId = InputNumber_I2`
  - **Erklärung**: Liest das VT-Eingabefeld `InputNumber_I2` und stellt den Index als bidirektionalen Plug `VALUEO` bereit; die Rückschreibung ins VT ist enthalten. Zusätzlich wird der Index unabhängig in `INI_AUI` persistiert (Schlüssel `KEY_I2_STORE`, Sektion `SECTION_I2_STORE`).
- **AOption_AUI_DEMUX_OptionalGroup_001**: Freigabe für Optionsgruppe 1
  - **Typ**: `Option::selection::AOption_AUI_DEMUX_1`
  - **Parameter**: keine
  - **Erklärung**: Gibt über `OUT1` die Optionsgruppe 1 frei (Socket `K`).
- **AOption_AUI_DEMUX_OptionalGroup_002**: Freigabe für Optionsgruppe 2
  - **Typ**: `Option::selection::AOption_AUI_DEMUX_1`
  - **Parameter**: keine
  - **Erklärung**: Gibt über `OUT1` die Optionsgruppe 2 frei (Socket `K`).
- **OptionalGroup_001**: Optionsgruppe 1
  - **Typ**: `MyLib_AX_Option::sys::OptionalGroup_001` (SubApp)
  - **Parameter**: keine
  - **Erklärung**: Wird über den Adapter `AOption_IN` freigegeben (QI anstelle Permit).
- **OptionalGroup_002**: Optionsgruppe 2
  - **Typ**: `MyLib_AX_Option::sys::OptionalGroup_002` (SubApp)
  - **Parameter**: keine
  - **Erklärung**: Wird über den Adapter `AOption_IN` freigegeben (QI anstelle Permit).

Programmablauf und Verbindungen

1. `NumericValue_TO_AUI2_AND_STORE_1.VALUEO` ⟶ `AOption_AUI_DEMUX_OptionalGroup_001.K`, danach `AOption_AUI_DEMUX_OptionalGroup_001.OUT1` ⟶ `OptionalGroup_001.AOption_IN` (Dropdown 1).
2. `NumericValue_TO_AUI2_AND_STORE_2.VALUEO` ⟶ `AOption_AUI_DEMUX_OptionalGroup_002.K`, danach `AOption_AUI_DEMUX_OptionalGroup_002.OUT1` ⟶ `OptionalGroup_002.AOption_IN` (Dropdown 2).

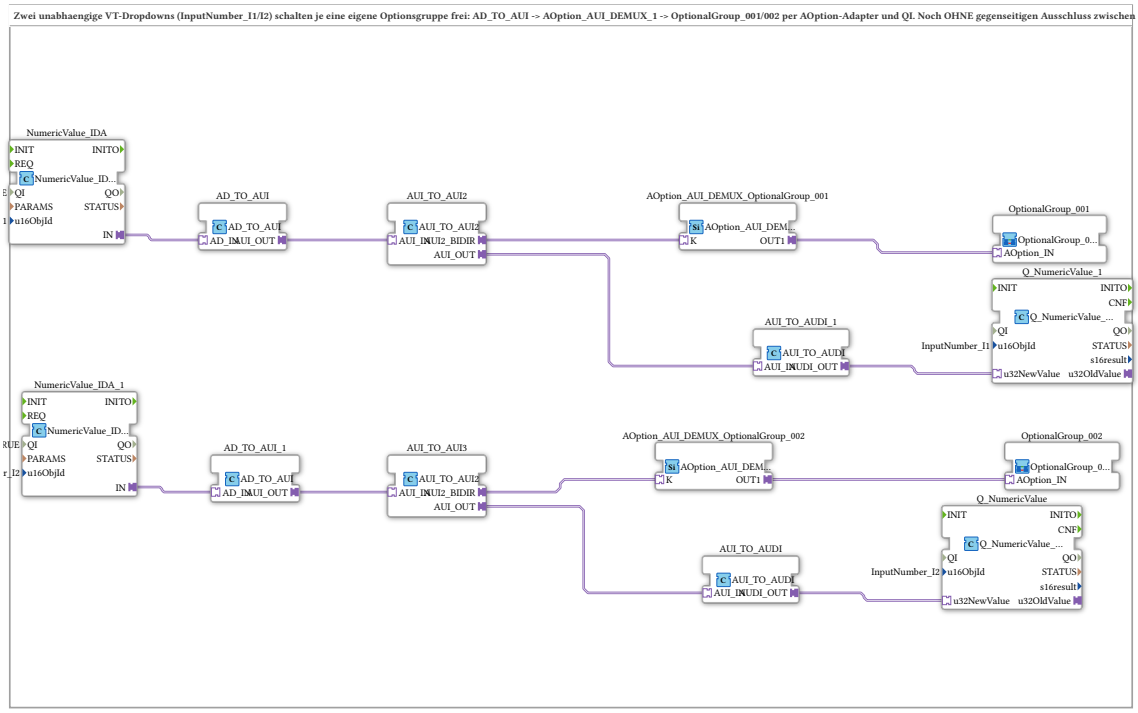
Zusammenfassung

Gegenüber `Uebung_094c2_AX` ändert sich nur der Bausteintyp und die zusätzlichen Parameter `KEY` und `SECTION`; die Verschaltung bleibt identisch.

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

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

2.1.2.416 Uebung_094c_AX: Zwei unabhängige VT-Dropdowns schalten je eine Optionsgruppe frei



• – ○ • – ○ • – ○ •

Einleitung

Zwei unabhängige VT-Dropdowns (`InputNumber_I1` und `InputNumber_I2`) schalten je eine eigene Optionsgruppe frei: `AD_TO_AUI` ⟶ `AOption_AUI_DEMUX_1` ⟶ `OptionalGroup_001`/`OptionalGroup_002` per `AOption`-Adapter und `QI`. Es gibt noch **keinen** gegenseitigen Ausschluss zwischen den beiden Gruppen; dafür wäre ein vorgeschalteter Arbiter nötig, der aus den zwei unabhängigen Dropdown-Events einen gemeinsamen K-Wert ableitet (offene Rückfrage aus Krauternter: Bunker_Klappen/Aufnahme_Rechts, „letzter Aktiviert gewinnt“ plus Boot-Tiebreak). Das Netzwerk enthält die Kette je Dropdown doppelt (inline dupliziert).

Verwendete Funktionsbausteine (FBs)

- **NumericValue_IDA**: VT-Dropdown 1 lesen
  - **Typ**: `isobus::UT::io::NumericValue::NumericValue_IDA`
  - **Parameter**: `QI = TRUE`, `u16ObjId = InputNumber_I1`
  - **Erklärung**: Liest das VT-Eingabefeld `InputNumber_I1`.

- **AD_TO_AUI**: Adapterwandler AD nach AUI
  - **Typ**: `adapter::conversion::unidirectional::AD_TO_AUI`
  - **Parameter**: keine
  - **Erklärung**: Wandelt `AD_IN` in `AUI_OUT`.
- **AUI_TO_AUI2**: Bidirektionale Brücke
  - **Typ**: `adapter::conversion::bidirectional::AUI_TO_AUI2`
  - **Parameter**: keine
  - **Erklärung**: Liefert den Index als `AUI2_BIDIR` an den Demultiplexer und gibt über `AUI_OUT` den zurückgemeldeten Wert aus.
- **AOption_AUI_DEMUX_OptionalGroup_001**: Freigabe für Optionsgruppe 1
  - **Typ**: `Option::selection::AOption_AUI_DEMUX_1`
  - **Parameter**: keine
  - **Erklärung**: Gibt über `OUT1` die Optionsgruppe 1 frei (Socket `K`).
- **OptionalGroup_001**: Optionsgruppe 1
  - **Typ**: `MyLib_AX_Option::sys::OptionalGroup_001` (SubApp)
  - **Parameter**: keine
  - **Erklärung**: Wird über den Adapter `AOption_IN` freigegeben (QI anstelle Permit).
- **AUI_TO_AUDI_1**: Adapterwandler AUI nach AUDI
  - **Typ**: `adapter::conversion::unidirectional::AUI_TO_AUDI`
  - **Parameter**: keine
  - **Erklärung**: Wandelt den zurückgemeldeten Wert für die VT-Rückschreibung.
- **Q_NumericValue_1**: VT-Wert schreiben
  - **Typ**: `isobus::UT::Q::Q_NumericValue_AUDI`
  - **Parameter**: `u16ObjId = InputNumber_I1`
  - **Erklärung**: Schreibt den Wert zurück in `InputNumber_I1`.
- **NumericValue_IDA_1**: VT-Dropdown 2 lesen
  - **Typ**: `isobus::UT::io::NumericValue::NumericValue_IDA`
  - **Parameter**: `QI = TRUE`, `u16ObjId = InputNumber_I2`
  - **Erklärung**: Liest das VT-Eingabefeld `InputNumber_I2`.
- **AD_TO_AUI_1**: Adapterwandler AD nach AUI
  - **Typ**: `adapter::conversion::unidirectional::AD_TO_AUI`
  - **Parameter**: keine
  - **Erklärung**: Wandelt `AD_IN` in `AUI_OUT`.
- **AUI_TO_AUI3**: Bidirektionale Brücke
  - **Typ**: `adapter::conversion::bidirectional::AUI_TO_AUI2`
  - **Parameter**: keine
  - **Erklärung**: Liefert den Index als `AUI2_BIDIR` an den Demultiplexer und gibt über `AUI_OUT` den zurückgemeldeten Wert aus.
- **AOption_AUI_DEMUX_OptionalGroup_002**: Freigabe für Optionsgruppe 2
  - **Typ**: `Option::selection::AOption_AUI_DEMUX_1`
  - **Parameter**: keine
  - **Erklärung**: Gibt über `OUT1` die Optionsgruppe 2 frei (Socket `K`).
- **OptionalGroup_002**: Optionsgruppe 2
  - **Typ**: `MyLib_AX_Option::sys::OptionalGroup_002` (SubApp)
  - **Parameter**: keine
  - **Erklärung**: Wird über den Adapter `AOption_IN` freigegeben (QI anstelle Permit).
- **AUI_TO_AUDI**: Adapterwandler AUI nach AUDI
  - **Typ**: `adapter::conversion::unidirectional::AUI_TO_AUDI`
  - **Parameter**: keine
  - **Erklärung**: Wandelt den zurückgemeldeten Wert für die VT-Rückschreibung.
- **Q_NumericValue**: VT-Wert schreiben
  - **Typ**: `isobus::UT::Q::Q_NumericValue_AUDI`
  - **Parameter**: `u16ObjId = InputNumber_I2`
  - **Erklärung**: Schreibt den Wert zurück in `InputNumber_I2`.

Programmablauf und Verbindungen

1. `NumericValue_IDA.IN` ⟶ `AD_TO_AUI.AD_IN` ⟶ `AUI_TO_AUI2.AUI_IN`; `AUI_TO_AUI2.AUI2_BIDIR` ⟶ `AOption_AUI_DEMUX_OptionalGroup_001.K`; `AOption_AUI_DEMUX_OptionalGroup_001.OUT1` ⟶ `OptionalGroup_001.AOption_IN` (Dropdown 1).

2. `NumericValue_IDA_1.IN` ⟶ `AD_TO_AUI_1.AD_IN` ⟶ `AUI_TO_AUI3.AUI_IN`; `AUI_TO_AUI3.AUI2_BIDIR` ⟶ `AOption_AUI_DEMUX_OptionalGroup_002.K`; `AOption_AUI_DEMUX_OptionalGroup_002.OUT1` ⟶ `OptionalGroup_002.AOption_IN` (Dropdown 2).
3. Echo Dropdown 1: `AUI_TO_AUI2.AUI_OUT` ⟶ `AUI_TO_AUDI_1.AUI_IN` ⟶ `Q_NumericValue_1.u32NewValue`.
4. Echo Dropdown 2: `AUI_TO_AUI3.AUI_OUT` ⟶ `AUI_TO_AUDI.AUI_IN` ⟶ `Q_NumericValue.u32NewValue`.

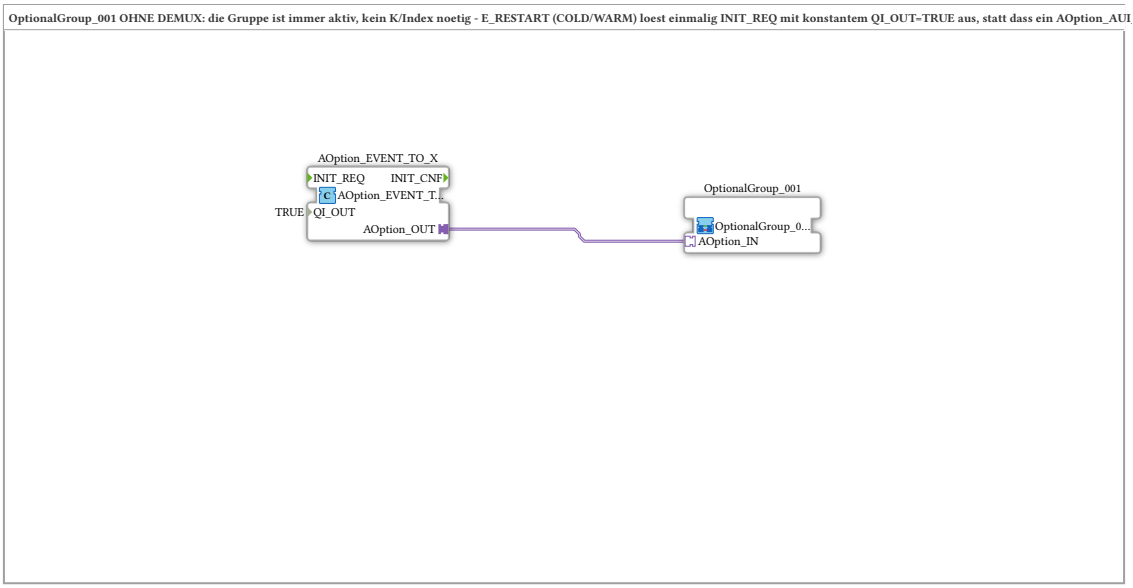
Zusammenfassung

Jedes Dropdown schaltet unabhängig seine eigene Optionsgruppe frei; ein gegenseitiger Ausschluss fehlt bewusst noch. Die doppelt vorhandene Kette wird in `Uebung_094c2_AX` in einen wiederverwendbaren Baustein ausgelagert.

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

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

2.1.2.417 Uebung_094d_AX: Optionsgruppe ohne DEMUX – immer aktiv, einmalig beim Ressourcenstart initialisiert



• – ○ • – ○ • – ○ •

Einleitung

Die Übung zeigt `OptionalGroup_001` OHNE DEMUX: Die Gruppe ist immer aktiv, es ist kein Index `K` nötig. Statt dass ein `AOption_AUI_DEMUX_N` je nach `K` einen von mehreren `OUTx`-Anschlüssen wählt, löst der Ressourcenstart einmalig `INIT_REQ` mit konstantem `QI_OUT = TRUE` aus (`E_RESTART` und `E_TRIG` liegen in der `EMB_RES`, nicht im SubApp). Der Baustein `AOption_EVENT_TO_X` hält dabei den Plug `AOption_OUT` des Adapters `AOption`.

Verwendete Funktionsbausteine (FBs)

- **AOption_EVENT_TO_X**: wandelt ein Event samt Qualifier in einen Options-Adapter
  - **Typ**: `Option::conversion::AOption_EVENT_TO_X`
  - **Parameter**: `QI_OUT = TRUE`
  - **Erklärung**: Composite-FB mit dem Plug `AOption_OUT`. `INIT_REQ` und `QI_OUT` werden an den Adapter weitergereicht, `INIT_CNF` des Adapters wird als `INIT_CNF` ausgegeben. `QI_OUT = TRUE` fordert die Option an bzw. aktiviert sie.
- **OptionalGroup_001**: Optionsgruppe 1
  - **Typ**: `MyLib_AX_Option::sys::OptionalGroup_001` (SubApp-Instanz)
  - **Parameter**: keine
  - **Erklärung**: Schaltet `Input_I1`/`Output_Q1` über den Options-Adapter (Socket `AOption_IN`) frei; `QI` aktiviert bzw. deaktiviert die ganze Gruppe.

Programmablauf und Verbindungen

1. In der EMB_RES führt E_RESTART (START.COLD) auf einen E_TRIG (EVENTTYPE = 'EInit'), der einmalig die Initialisierung auslöst, die als AOption_EVENT_TO_X.INIT_REQ ankommt; der Parameter QI_OUT = TRUE ist konstant gesetzt.
2. AOption_EVENT_TO_X.AOption_OUT → OptionalGroup_001.AOption_IN (AdapterConnection): Der Adapter reicht INIT_REQ und QI_OUT = TRUE an die Gruppe weiter.
3. Die Gruppe initialisiert ihre I/O-Bausteine mit QI = TRUE und ist damit dauerhaft aktiv.

Verhalten: Es gibt keinen DEMUX und keinen Index K. Die Gruppe wird nach dem Start einmal initialisiert und bleibt aktiv.

Lernziele

- Eine Optionsgruppe auch ohne Auswahl-DEMUX betreiben
- Den Options-Adapter AOption mit AOption_EVENT_TO_X direkt aus einem Event speisen
- Die Initialisierung einmalig über E_RESTART und E_TRIG in der EMB_RES anstoßen
- Den Qualifier QI_OUT als Konstante setzen

Zusammenfassung

Gegenüber den Übungen mit AOption_AUI_DEMUX_N entfallen Index und Auswahl: AOption_EVENT_TO_X mit QI_OUT = TRUE speist die Gruppe OptionalGroup_001 direkt, sodass sie immer aktiv ist.

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

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

2.1.2.418 Uebung_095_AX: Beispiel für AX_SELECT



Dieser Artikel beschreibt die logiBUS®-Übung Uebung_095_AX.

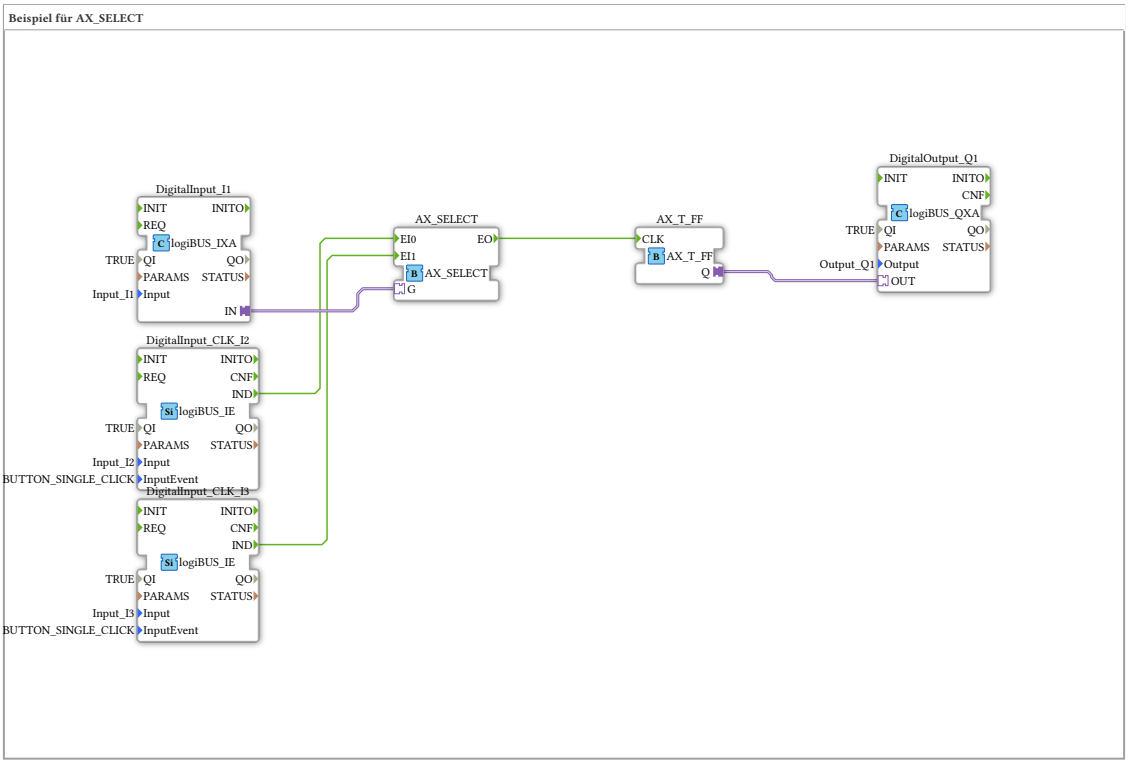
Ziel der Übung

Auswahl einer Event-Quelle (Gegenteil von E_SPLIT oder E_SWITCH).

Beschreibung und Komponenten

Die Subapplikation Uebung_095_AX.SUB nutzt einen AX_SELECT Baustein.

Funktionsbausteine (FBs)



- **`I1`**: Wahlschalter (Gate `G`).
- **`I2`**: Event-Quelle A.
- **`I3`**: Event-Quelle B.
- **`E_SELECT`**: Leitet entweder A oder B an den Ausgang weiter.

Funktionsweise

- Ist `I1` aus (`G=FALSE`), werden Events von `I2` (`EI0`) zum Ausgang durchgelassen. Events von `I3` werden ignoriert.
- Ist `I1` an (`G=TRUE`), werden Events von `I3` (`EI1`) zum Ausgang durchgelassen. Events von `I2` werden ignoriert.

Der Ausgang triggert ein Flip-Flop (`Q1`). Man kann also wählen, *welcher* Taster das Licht schalten darf.

Anwendungsbeispiel

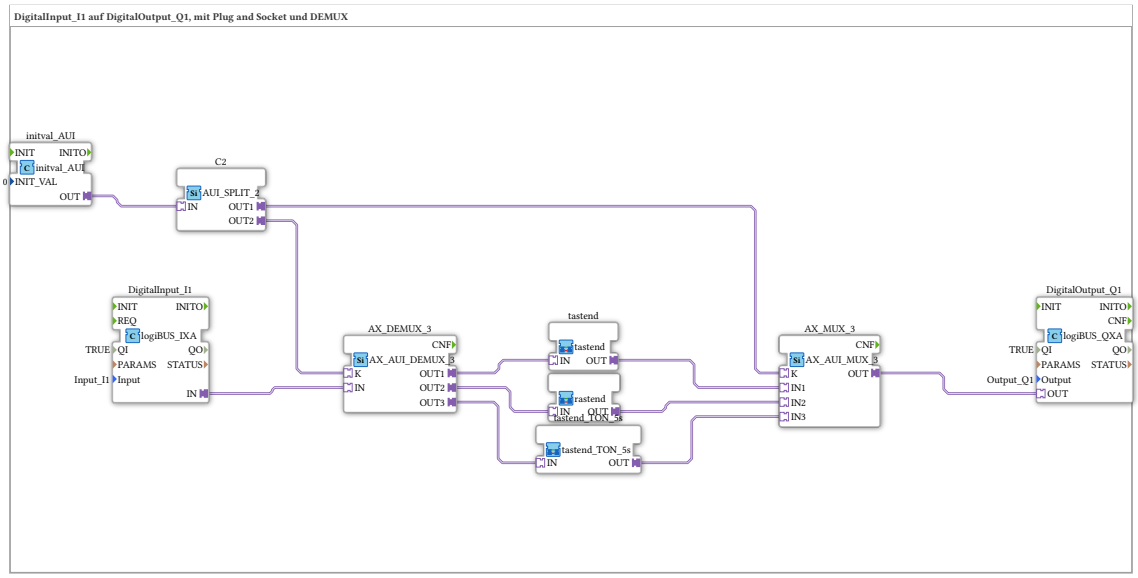
Bedienplatz-Umschaltung: Ein Schlüsselschalter entscheidet, ob die Maschine vom Hauptpult (`I2`) oder vom Wartungspult (`I3`) gesteuert wird.

2.1.2.419 Uebung_103

Hinweis

Diese Übung existiert in den aktuellen Trainingsquellen nicht mehr (entfernt, zusammengelegt oder umbenannt). Diese Seite bleibt nur als Platzhalter für bestehende Verweise erhalten. Die aktuellen Übungen stehen in der Übungsübersicht.

2.1.2.420 Uebung_103c: DigitalInput_I1 auf DigitalOutput_Q1, mit Plug and Socket und DEMUX



• – ○ • – ○ • – ○ •



Einleitung

Ziel der Übung ist das Testen eines spezifischen Pfads der MUX/DEMUX-Struktur. Im Vergleich zu Uebung_103 wurde das VT-Eingabefeld entfernt: Der Auswahlindex für `AX_DEMUX_3` und `AX_MUX_3` wird stattdessen von einem `initval_AUI` mit festem Wert vorgegeben. Der Taster `I1` wirkt dadurch dauerhaft über denselben der drei Zweige (`tastend`, `rastend`, `tastend_TON_5s`) auf den Ausgang `Q1`, obwohl die Struktur für die anderen Modi noch vorhanden ist. Das dient oft zum Debugging oder zum schnellen Einfrieren einer Konfiguration.

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_I1**: logiBUS Digitaleingang
  - **Typ**: `logiBUS::io::DI::logiBUS_IXA`
  - **Parameter**: `QI = TRUE`, `Input = Input_I1`
  - **Erklärung**: Liefert den Zustand von `I1` als AX-Adaptersignal.
- **initval_AUI**: fester Auswahlindex
  - **Typ**: `adapter::types::unidirectional::AUI::initval::initval_AUI`
  - **Parameter**: `INIT_VAL = 0`
  - **Erklärung**: Stellt den konstanten Index 0 als AUI-Signal bereit.
- **C2**: Verteiler des Index
  - **Typ**: `adapter::events::unidirectional::AUI_SPLIT_2`
  - **Parameter**: keine
  - **Erklärung**: Verteilt den Index auf `AX_AUI_MUX_3.K` (`OUT1`) und `AX_AUI_DEMUX_3.K` (`OUT2`).
- **AX_DEMUX_3**: Demultiplexer
  - **Typ**: `adapter::selection::unidirectional::AX_AUI_DEMUX_3`
  - **Parameter**: keine
  - **Erklärung**: Leitet das Signal von `IN` an einen der Ausgänge `OUT1` bis `OUT3` weiter; der Index kommt über den Adapter `K` und ist 0-basiert (Index 0 ⟶ `OUT1`, 1 ⟶ `OUT2`, 2 ⟶ `OUT3`).
- **tastend**: Zweig 1, tastende Funktion
  - **Typ**: `MyLib::sys::tastend` (SubApp-Instanz, siehe tastend)
  - **Parameter**: keine
  - **Erklärung**: Leitet das Signal direkt (1:1) von `IN` an `OUT` weiter.
- **rastend**: Zweig 2, rastende Funktion

  - **Typ**: `MyLib::sys::rastend` (SubApp-Instanz, siehe rastend)
  - **Parameter**: keine
  - **Erklärung**: Wandelt den Taster in einen Schalter (Toggle) um; das Umschalten erfolgt beim Loslassen (fallende Flanke über `AX_F_TRIG` auf `AX_T_FF`).
- **tastend_TON_5s**: Zweig 3, tastend mit 5 s Einschaltverzögerung
  - **Typ**: `MyLib::sys::tastend_TON_5s` (SubApp-Instanz, siehe tastend_TON_5s)
  - **Parameter**: keine
  - **Erklärung**: Leitet das Signal über einen `AX_TON` mit `PT = t#5s` weiter, also mit 5 s Einschaltverzögerung.
- **AX_MUX_3**: Multiplexer
  - **Typ**: `adapter::selection::unidirectional::AX_AUI_MUX_3`
  - **Parameter**: keine
  - **Erklärung**: Gibt den durch den Index `K` gewählten Eingang `IN1` bis `IN3` an `OUT` weiter (Index 0 ⟶ `IN1`, 1 ⟶ `IN2`, 2 ⟶ `IN3`).
- **DigitalOutput_Q1**: logiBUS Digitalausgang
  - **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
  - **Parameter**: `QI = TRUE`, `Output = Output_Q1`
  - **Erklärung**: Gibt das Ergebnis am physischen Ausgang `Q1` aus.

Programmablauf und Verbindungen

1. `initval_AUI.OUT` ⟶ `C2.IN`; `C2.OUT2` ⟶ `AX_DEMUX_3.K` und `C2.OUT1` ⟶ `AX_MUX_3.K`: Beide Selektoren erhalten denselben festen Index.
2. `DigitalInput_I1.IN` ⟶ `AX_DEMUX_3.IN`: Der Taster speist den Demultiplexer.
3. `AX_DEMUX_3.OUT1` ⟶ `tastend.IN`, `AX_DEMUX_3.OUT2` ⟶ `rastend.IN`, `AX_DEMUX_3.OUT3` ⟶ `tastend_TON_5s.IN`: Die drei Zweige.
4. `tastend.OUT` ⟶ `AX_MUX_3.IN1`, `rastend.OUT` ⟶ `AX_MUX_3.IN2`, `tastend_TON_5s.OUT` ⟶ `AX_MUX_3.IN3`: Die Zweige laufen im Multiplexer zusammen.
5. `AX_MUX_3.OUT` ⟶ `DigitalOutput_Q1.OUT`: Das Ergebnis des gewählten Zweigs wird auf `Q1` gelegt.

Verhalten: Mit `INIT_VAL = 0` ist der Index 0 gewählt, also der Zweig `tastend`: `Q1` folgt `I1` direkt. Wird der Wert von `INIT_VAL` im Quelltext auf 1 bzw. 2 geändert, wirkt stattdessen der Zweig `rastend` bzw. `tastend_TON_5s`.

Anwendungsbeispiel

Fest eingestellte Konfiguration einer umschaltbaren Bedienlogik, zum Debugging eines einzelnen Zweigs oder um eine Einstellung dauerhaft einzufrieren.

Zusammenfassung

Die Übung `Uebung_103c` zeigt die MUX/DEMUX-Struktur mit festem Auswahlindex: Ein `initval_AUI` legt über `AUI_SPLIT_2` denselben Index an `AX_AUI_DEMUX_3` und `AX_AUI_MUX_3` an, sodass nur ein Zweig der drei Verarbeitungslogiken aktiv ist.

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

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

2.1.2.421 Uebung_103c2: DigitalInput_I1 auf DigitalOutput_Q1, mit Plug and Socket und DEMUX



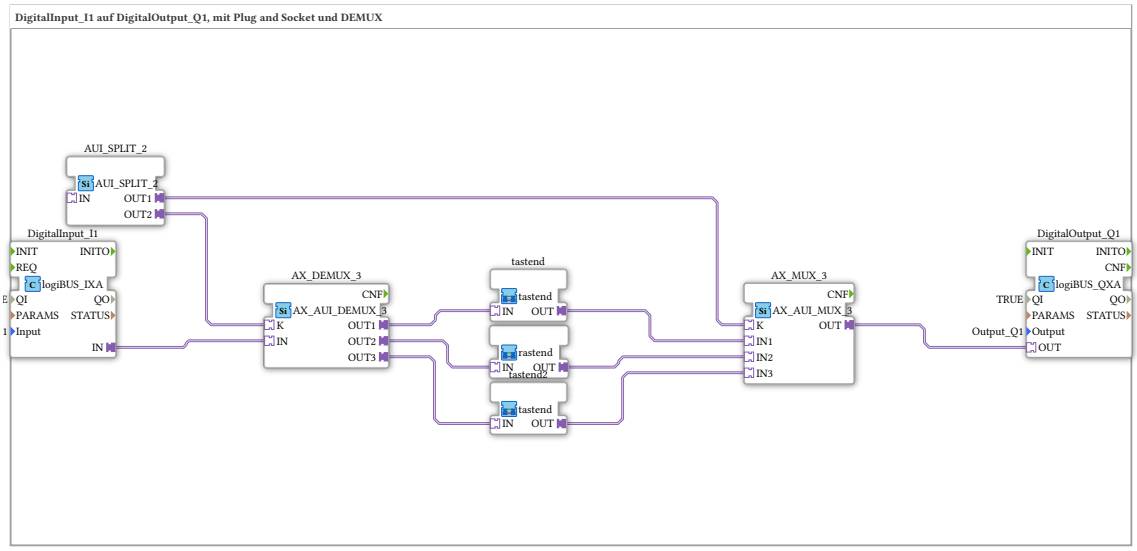
Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_103c2`.

Ziel der Übung

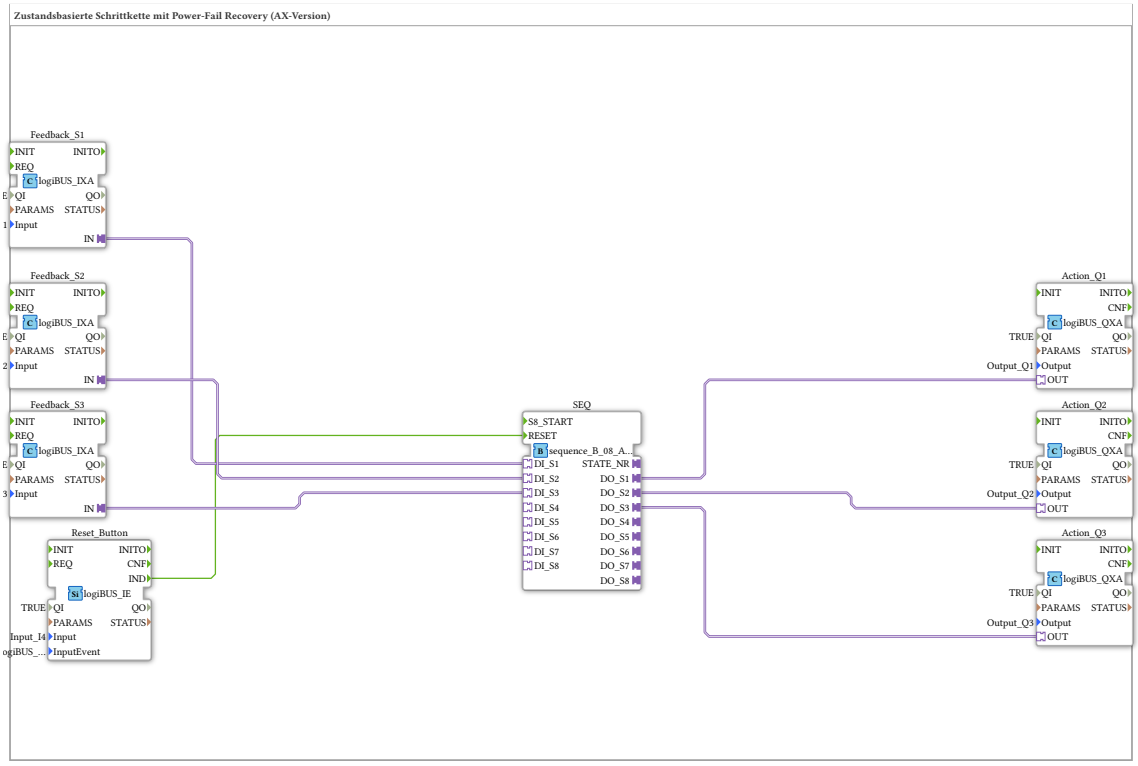
Variation der Modus-Selektion.

Beschreibung

Eine weitere Variante der MUX/DEMUX Übung. Hier wird die interne Verschaltung der Pfade (z.B. `tastend2`) leicht variiert, um unterschiedliche Kombinationen zu testen.



2.1.2.422 Uebung_104_AX: Zustandsbasierte Schrittkette mit Power-Fail Recovery (AX-Version)



• — ○ • — ○ • — ○ •

Einleitung

Diese Übung realisiert eine **zustandsbasierte Schrittkette mit Power-Fail Recovery** für die AX-Variante (Adapter-basiert). Das System kann nach einem Spannungsausfall oder Neustart selbstständig in den Zustand zurückkehren, dessen zugehöriger Eingangssensor gerade ein TRUE-Signal liefert. Der verwendete Sequenzbaustein `sequence_B_08_AX_AX` bildet das Kernstück und wird über Adapter mit den Ein- und Ausgangsbausteinen verbunden.

Verwendete Funktionsbausteine (FBs)

Sub-Baustein: SEQ
- **Typ**: `logiBUS::utils::sequence::boolean::sequence_B_08_AX_AX`
- **Parameter**: Keine expliziten Parameter gesetzt.
- **Ereigniseingänge**:
  - `RESET` – verbunden mit dem Reset-Taster (Reset_Button.IND)
- **Adaptereingänge**:
  - `DI_S1`, `DI_S2`, `DI_S3` – jeweils verbunden mit den Feedback-Adaptern
- **Adapterausgänge**:
  - `DO_S1`, `DO_S2`, `DO_S3` – jeweils verbunden mit den Aktions-Adaptern
- **Funktionsweise**: Eine Sequenzsteuerung für bis zu acht Schritte, die auf booleschen Adapterwerten basiert. Beim Einschalten oder nach einem Reset springt die Sequenz direkt in den Schritt, dessen zugehöriger Eingangsadapter TRUE liefert (Power-Fail Recovery). Während des normalen Betriebs durchläuft sie eine festgelegte Schrittfolge.

Sub-Baustein: Feedback_S1
- **Typ**: `logiBUS::io::DI::logiBUS_IXA`
- **Parameter**:
  - `QI` = `TRUE`
  - `Input` = `Input_I1`
- **Adapterausgang**: `IN` – verbunden mit `SEQ.DI_S1`
- **Funktionsweise**: Digitaler Eingangsadapter, der den physikalischen Sensor `Input_I1` in einen logischen Adapter umwandelt. Bei `QI=TRUE` ist der Adapter aktiv.

Sub-Baustein: Feedback_S2
- **Typ**: `logiBUS::io::DI::logiBUS_IXA`
- **Parameter**:
  - `QI` = `TRUE`
  - `Input` = `Input_I2`
- **Adapterausgang**: `IN` – verbunden mit `SEQ.DI_S2`
- **Funktionsweise**: Analog zu Feedback_S1 für Sensor `Input_I2`.

Sub-Baustein: Feedback_S3
- **Typ**: `logiBUS::io::DI::logiBUS_IXA`
- **Parameter**:
  - `QI` = `TRUE`
  - `Input` = `Input_I3`
- **Adapterausgang**: `IN` – verbunden mit `SEQ.DI_S3`
- **Funktionsweise**: Analog zu Feedback_S1 für Sensor `Input_I3`.

Sub-Baustein: Reset_Button
- **Typ**: `logiBUS::io::DI::logiBUS_IE`
- **Parameter**:
  - `QI` = `TRUE`
  - `Input` = `Input_I4`
  - `InputEvent` = `logiBUS::io::DI::logiBUS_DI_Events::BUTTON_SINGLE_CLICK`
- **Ereignisausgang**: `IND` – verbunden mit `SEQ.RESET`
- **Funktionsweise**: Erkennt einen einzelnen Tastendruck (Single-Click) auf den physischen Eingang `Input_I4` und erzeugt ein Ereignis am Ausgang `IND`, das die Sequenz zurücksetzt.

Sub-Baustein: Action_Q1
- **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
- **Parameter**:
  - `QI` = `TRUE`
  - `Output` = `Output_Q1`
- **Adaptereingang**: `OUT` – verbunden mit `SEQ.DO_S1`
- **Funktionsweise**: Digitaler Ausgangsadapter, der den logischen Adapterwert auf den physikalischen Ausgang `Output_Q1` ausgibt.

Sub-Baustein: Action_Q2

- **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
- **Parameter**:
  - `QI` = `TRUE`
  - `Output` = `Output_Q2`
- **Adaptereingang**: `OUT` – verbunden mit `SEQ.DO_S2`
- **Funktionsweise**: Analog zu Action_Q1 für Ausgang `Output_Q2`.

Sub-Baustein: Action_Q3

- **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
- **Parameter**:
  - `QI` = `TRUE`
  - `Output` = `Output_Q3`
- **Adaptereingang**: `OUT` – verbunden mit `SEQ.DO_S3`
- **Funktionsweise**: Analog zu Action_Q1 für Ausgang `Output_Q3`.

Programmablauf und Verbindungen

Die **Schrittkette** (SEQ) verfügt über drei Eingangsadapter (`DI_S1` … `DI_S3`) und drei Ausgangsadapter (`DO_S1` … `DO_S3`). Die Eingänge sind über die Feedback-Bausteine mit den physikalischen Sensoren `Input_I1` … `Input_I3` verbunden. Die Ausgänge steuern über die Action-Bausteine die Aktoren `Output_Q1` … `Output_Q3` an.

Reset-Verhalten: Drücken des Tasters an `Input_I4` erzeugt ein Ereignis (`IND`), das den RESET-Eingang der Sequenz triggert. Daraufhin springt die Sequenz in den Zustand, dessen zugehöriger Eingangsadapter TRUE führt (z. B. wenn Sensor 2 aktiv ist, wird Schritt 2 gestartet). Dieses Verhalten stellt das **Power-Fail Recovery** dar: Nach einem Stromausfall oder Neustart kehrt die Steuerung automatisch in einen definierten Zustand zurück.

Normalbetrieb: Im normalen Ablauf durchläuft die Sequenz eine vordefinierte Schrittfolge (nicht im XML dargestellt, da die Schritte im FB `sequence_B_08_AX_AX` konfiguriert werden).

Lernziele

- Verständnis von zustandsbasierten Steuerungen mit Recovery-Mechanismus
- Umgang mit IEC 61499-Adaptern in der 4diac-IDE
- Einsatz von logiBUS I/O-Bausteinen (Eingangs-/Ausgangsadapter)
- Analyse und Fehlerbehandlung bei Spannungsausfällen

Schwierigkeitsgrad

Mittel – Grundkenntnisse in 4diac und Schrittketten sind erforderlich.

Vorkenntnisse

- Grundlagen der IEC 61499
- Bedienung der 4diac-IDE
- Vertrautheit mit logiBUS-I/O (Sensoren, Aktoren)

Starten der Übung

1. Öffnen Sie die Übung `Uebung_104_AX` in der 4diac-IDE.
2. Verbinden Sie die physikalischen Ein-/Ausgänge:
   - `Input_I1`, `I2`, `I3` – z. B. Endlagenschalter oder Sensoren
   - `Input_I4` – Taster für Reset
   - `Output_Q1`, `Q2`, `Q3` – z. B. Ventile oder Motoren
3. Laden Sie die Applikation auf die Zielhardware (z. B. logiBUS-Steuerung).
4. Testen Sie das Power-Fail Recovery: Unterbrechen und Wiederherstellen der Spannungsversorgung – die Maschine sollte in den durch die Sensoren vorgegebenen Zustand zurückkehren.

Zusammenfassung

Die Übung **Uebung_104_AX** demonstriert eine **zustandsbasierte Schrittkette mit Power-Fail Recovery** mithilfe des Sequenzbausteins `sequence_B_08_AX_AX` in Verbindung mit logiBUS I/O-Adaptern. Durch die adapterbasierte Verbindung werden Sensoren und Aktoren modular eingebunden. Das Recovery-Verhalten sorgt dafür, dass die Steuerung nach einem Spannungsausfall automatisch den zum

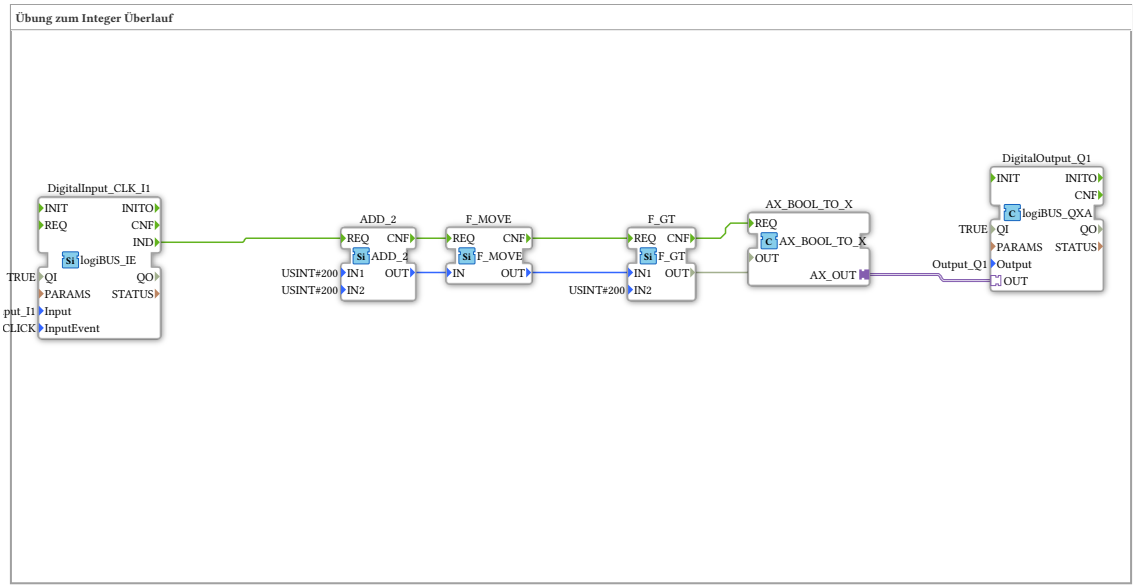
aktuellen Sensorzustand passenden Schritt einnimmt – eine wichtige Eigenschaft für sicherheitskritische oder unterbrechungsempfindliche Anwendungen.

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

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

2.1.2.423 Uebung_110_AX: Übung zum Integer Überlauf

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_110_AX (Übung zum Integer Überlauf).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Übung zum Integer Überlauf**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_110_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
- **ADD_2**: Instanz des Typs iec61131::arithmetic::ADD_2.
  - Parameter IN1 = USINT#200
  - Parameter IN2 = USINT#200
- **DigitalInput_CLK_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **F_GT**: Instanz des Typs iec61131::comparison::F_GT.
  - Parameter IN2 = USINT#200
- **F_MOVE**: Instanz des Typs iec61131::selection::F_MOVE.
- **AX_BOOL_TO_X**: Instanz des Typs adapter::conversion::unidirectional::AX_BOOL_TO_X.

Verbindungen und Schnittstellen

Adapterverbindungen:

- AX_BOOL_TO_X.AX_OUT -> DigitalOutput_Q1.OUT

Ereignisverbindungen:

- DigitalInput_CLK_I1.IND -> ADD_2.REQ
- F_MOVE.CNF -> F_GT.REQ
- F_GT.CNF -> AX_BOOL_TO_X.REQ
- ADD_2.CNF -> F_MOVE.REQ

Datenverbindungen:

- F_MOVE.OUT -> F_GT.IN1
- F_GT.OUT -> AX_BOOL_TO_X.OUT
- ADD_2.OUT -> F_MOVE.IN

Hinweise aus dem Modell

Arithmetischer Überlauf

https://de.wikipedia.org/wiki/Arithmetischer_%C3%9Cberlauf
https://www.youtube.com/watch?v=TLanGc-c9Ww

Programmablauf und Funktionsweise

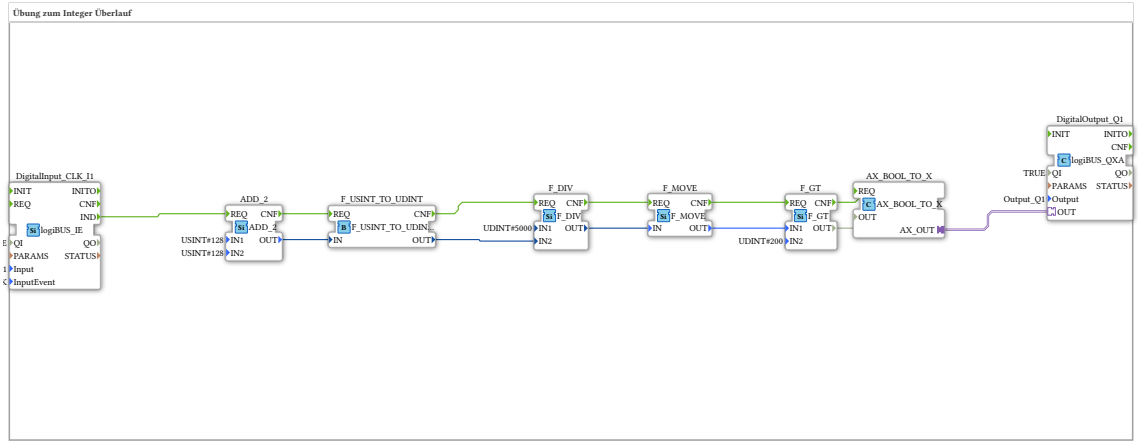
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_110_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.424 Uebung_111_AX: Übung zum Integer Überlauf

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_111_AX (Übung zum Integer Überlauf).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Übung zum Integer Überlauf**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_111_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1

- **ADD_2**: Instanz des Typs iec61131::arithmetic::ADD_2.
  - Parameter IN1 = USINT#128
  - Parameter IN2 = USINT#128
- **DigitalInput_CLK_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **F_GT**: Instanz des Typs iec61131::comparison::F_GT.
  - Parameter IN2 = UDINT#200
- **F_DIV**: Instanz des Typs iec61131::arithmetic::F_DIV.
  - Parameter IN1 = UDINT#5000
- **F_USINT_TO_UDINT**: Instanz des Typs iec61131::conversion::F_USINT_TO_UDINT.
- **F_MOVE**: Instanz des Typs iec61131::selection::F_MOVE.
- **AX_BOOL_TO_X**: Instanz des Typs adapter::conversion::unidirectional::AX_BOOL_TO_X.

Verbindungen und Schnittstellen

Adapterverbindungen:

- AX_BOOL_TO_X.AX_OUT -> DigitalOutput_Q1.OUT

Ereignisverbindungen:

- DigitalInput_CLK_I1.IND -> ADD_2.REQ
- F_GT.CNF -> AX_BOOL_TO_X.REQ
- ADD_2.CNF -> F_USINT_TO_UDINT.REQ
- F_USINT_TO_UDINT.CNF -> F_DIV.REQ
- F_DIV.CNF -> F_MOVE.REQ
- F_MOVE.CNF -> F_GT.REQ

Datenverbindungen:

- F_GT.OUT -> AX_BOOL_TO_X.OUT
- ADD_2.OUT -> F_USINT_TO_UDINT.IN
- F_USINT_TO_UDINT.OUT -> F_DIV.IN2
- F_MOVE.OUT -> F_GT.IN1
- F_DIV.OUT -> F_MOVE.IN

Programmablauf und Funktionsweise

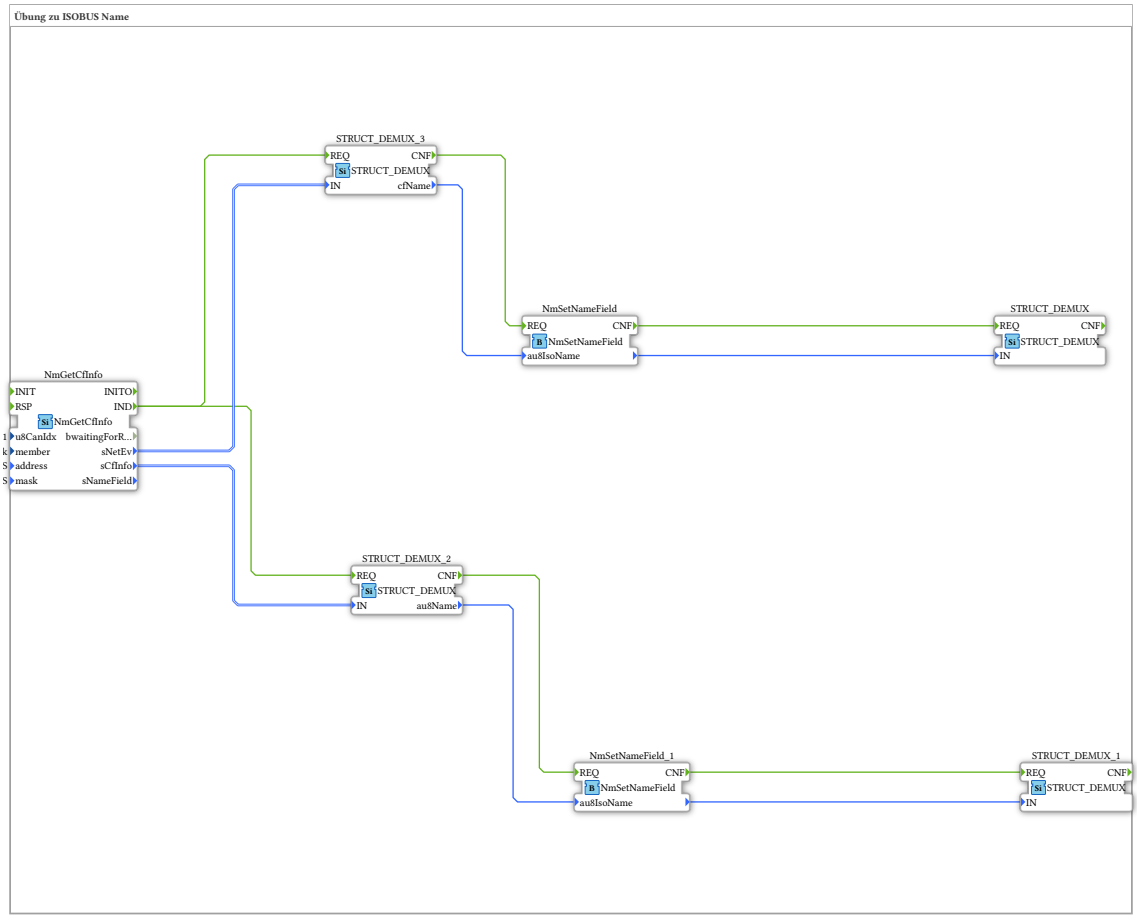
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_111_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.425 Uebung_120_AX: Übung zu ISOBUS Name

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_120_AX (Übung zu ISOBUS Name).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Übung zu ISOBUS Name**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_120_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **NmGetCfInfo**: Instanz des Typs isobus::pgn::NmGetCfInfo.
  - Parameter u8CanIdx = NODE1
  - Parameter member = network
  - Parameter address = ADD_ALL_PASS
  - Parameter mask = FLT_ALL_PASS
- **STRUCT_DEMUX_2**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **STRUCT_DEMUX_3**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **NmSetNameField**: Instanz des Typs isobus::pgn::NmSetNameField.
- **NmSetNameField_1**: Instanz des Typs isobus::pgn::NmSetNameField.
- **STRUCT_DEMUX**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **STRUCT_DEMUX_1**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.

Verbindungen und Schnittstellen

Ereignisverbindungen:

- NmGetCfInfo.IND -> STRUCT_DEMUX_3.REQ
- NmGetCfInfo.IND -> STRUCT_DEMUX_2.REQ
- STRUCT_DEMUX_3.CNF -> NmSetNameField.REQ
- STRUCT_DEMUX_2.CNF -> NmSetNameField_1.REQ
- NmSetNameField.CNF -> STRUCT_DEMUX.REQ

- NmSetNameField_1.CNF -> STRUCT_DEMUX_1.REQ

Datenverbindungen:

- NmGetCfInfo.sNetEv -> STRUCT_DEMUX_3.IN
- NmGetCfInfo.sCfInfo -> STRUCT_DEMUX_2.IN
- STRUCT_DEMUX_3.cfName -> NmSetNameField.au8IsoName
- STRUCT_DEMUX_2.au8Name -> NmSetNameField_1.au8IsoName
- NmSetNameField. -> STRUCT_DEMUX.IN
- NmSetNameField_1. -> STRUCT_DEMUX_1.IN

Hinweise aus dem Modell

mit einem Event an RSP wird der nächste ACL abgefragt.

Programmablauf und Funktionsweise

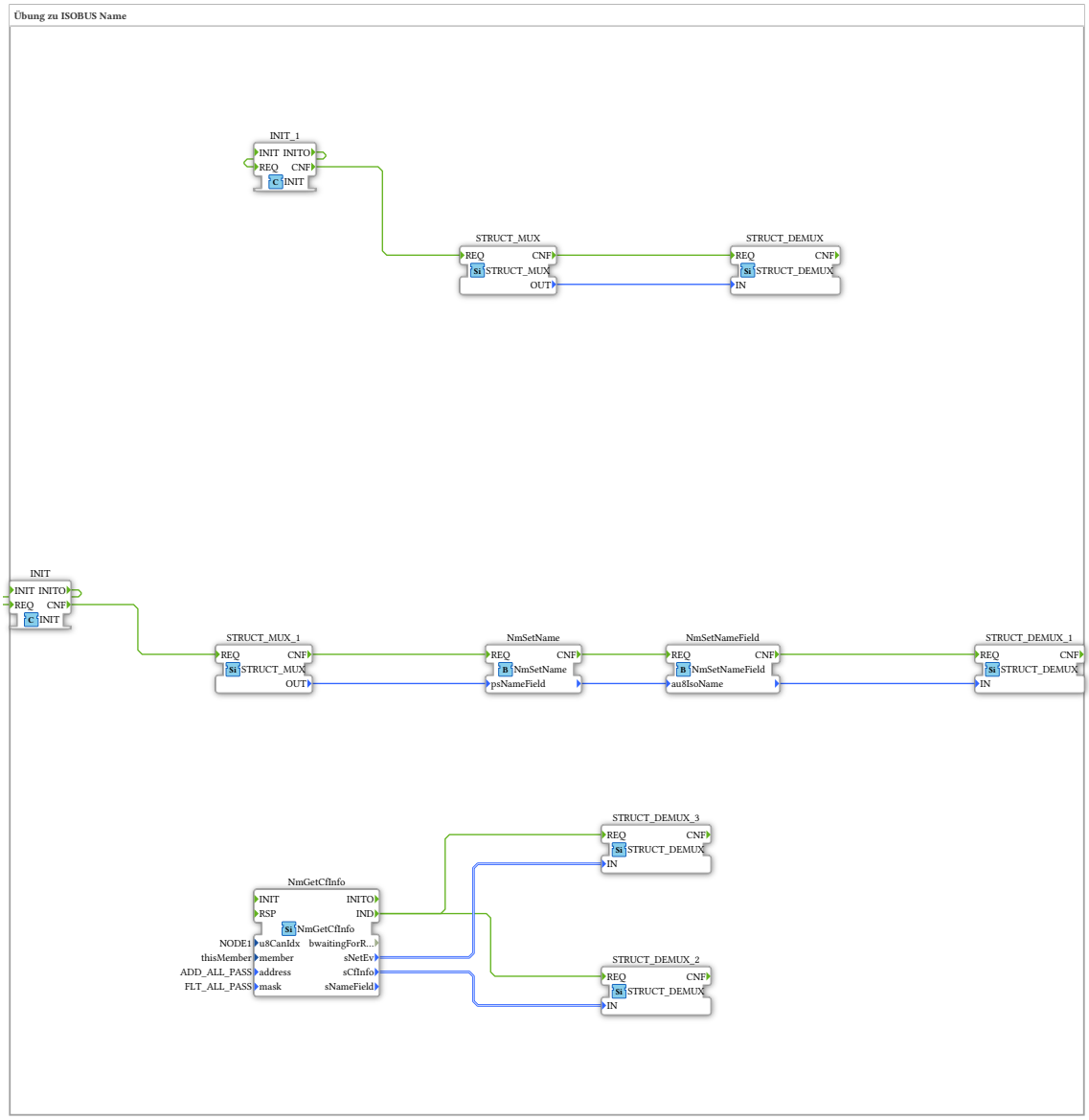
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_120_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.426 Uebung_121_AX: Übung zu ISOBUS Name

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_121_AX (Übung zu ISOBUS Name).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Übung zu ISOBUS Name**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_121_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **STRUCT_DEMUX**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **STRUCT_MUX**: Instanz des Typs eclipse4diac::convert::STRUCT_MUX.
- **NmSetName**: Instanz des Typs isobus::pgn::NmSetName.
- **NmSetNameField**: Instanz des Typs isobus::pgn::NmSetNameField.
- **STRUCT_MUX_1**: Instanz des Typs eclipse4diac::convert::STRUCT_MUX.
- **STRUCT_DEMUX_1**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **NmGetCfInfo**: Instanz des Typs isobus::pgn::NmGetCfInfo.
  - Parameter u8CanIdx = NODE1
  - Parameter member = thisMember
  - Parameter address = ADD_ALL_PASS
  - Parameter mask = FLT_ALL_PASS

- **STRUCT_DEMUX_2**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **STRUCT_DEMUX_3**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **INIT**: Instanz des Typs iec61131::booleanOperators::INIT.
- **INIT_1**: Instanz des Typs iec61131::booleanOperators::INIT.

Verbindungen und Schnittstellen

Ereignisverbindungen:

- STRUCT_MUX.CNF -> STRUCT_DEMUX.REQ
- NmSetName.CNF -> NmSetNameField.REQ
- STRUCT_MUX_1.CNF -> NmSetName.REQ
- NmSetNameField.CNF -> STRUCT_DEMUX_1.REQ
- NmGetCfInfo.IND -> STRUCT_DEMUX_3.REQ
- NmGetCfInfo.IND -> STRUCT_DEMUX_2.REQ
- INIT.INITO -> INIT.REQ
- INIT.CNF -> STRUCT_MUX_1.REQ
- INIT_1.INITO -> INIT_1.REQ
- INIT_1.CNF -> STRUCT_MUX.REQ

Datenverbindungen:

- STRUCT_MUX.OUT -> STRUCT_DEMUX.IN
- STRUCT_MUX_1.OUT -> NmSetName.psNameField
- NmSetName. -> NmSetNameField.au8IsoName
- NmSetNameField. -> STRUCT_DEMUX_1.IN
- NmGetCfInfo.sNetEv -> STRUCT_DEMUX_3.IN
- NmGetCfInfo.sCfInfo -> STRUCT_DEMUX_2.IN

Programmablauf und Funktionsweise

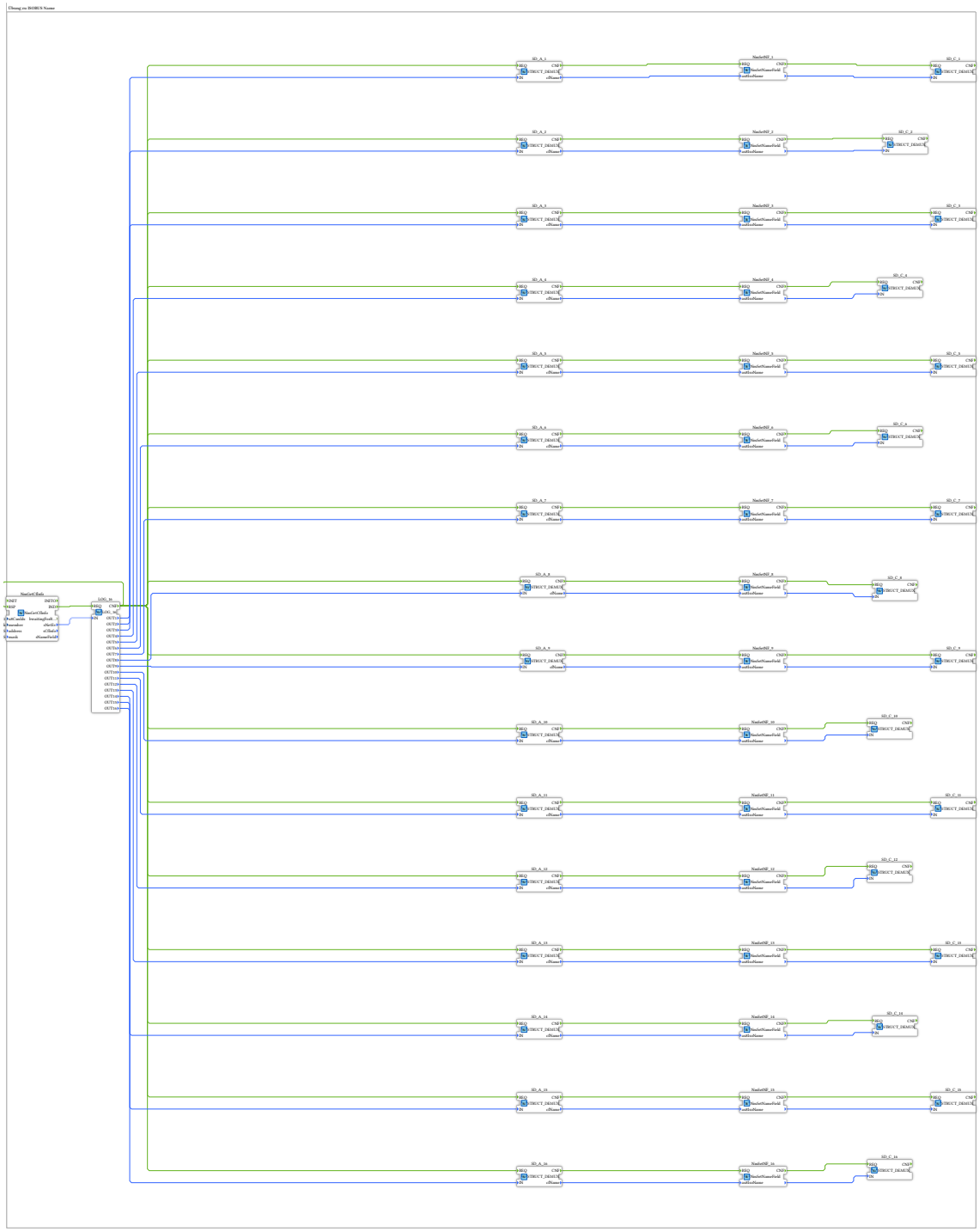
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_121_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.427 Uebung_122_AX: Übung zu ISOBUS Name

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_122_AX (Übung zu ISOBUS Name).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Übung zu ISOBUS Name**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_122_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **NmGetCfInfo**: Instanz des Typs isobus::pgn::NmGetCfInfo.
  - Parameter u8CanIdx = NODE1
  - Parameter member = network

  - Parameter address = ADD_ALL_PASS
  - Parameter mask = FLT_ALL_PASS
- **LOG_16**: Instanz des Typs logiBUS::utils::logging::LOG_16.
- **SD_A_1**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_A_2**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_A_3**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_A_4**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_A_5**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_A_6**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_A_7**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_A_8**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_A_9**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_A_10**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_A_11**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_A_12**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_A_13**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_A_14**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_A_15**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_A_16**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **NmSetNF_1**: Instanz des Typs isobus::pgn::NmSetNameField.
- **NmSetNF_2**: Instanz des Typs isobus::pgn::NmSetNameField.
- **NmSetNF_3**: Instanz des Typs isobus::pgn::NmSetNameField.
- **NmSetNF_4**: Instanz des Typs isobus::pgn::NmSetNameField.
- **NmSetNF_5**: Instanz des Typs isobus::pgn::NmSetNameField.
- **NmSetNF_6**: Instanz des Typs isobus::pgn::NmSetNameField.
- **NmSetNF_7**: Instanz des Typs isobus::pgn::NmSetNameField.
- **NmSetNF_8**: Instanz des Typs isobus::pgn::NmSetNameField.
- **NmSetNF_9**: Instanz des Typs isobus::pgn::NmSetNameField.
- **NmSetNF_10**: Instanz des Typs isobus::pgn::NmSetNameField.
- **NmSetNF_11**: Instanz des Typs isobus::pgn::NmSetNameField.
- **NmSetNF_12**: Instanz des Typs isobus::pgn::NmSetNameField.
- **NmSetNF_13**: Instanz des Typs isobus::pgn::NmSetNameField.
- **NmSetNF_14**: Instanz des Typs isobus::pgn::NmSetNameField.
- **NmSetNF_15**: Instanz des Typs isobus::pgn::NmSetNameField.
- **NmSetNF_16**: Instanz des Typs isobus::pgn::NmSetNameField.
- **SD_C_1**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_C_2**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_C_3**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_C_4**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_C_5**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_C_6**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_C_7**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_C_8**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_C_9**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_C_10**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_C_11**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_C_12**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_C_13**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_C_14**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_C_15**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_C_16**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.

Verbindungen und Schnittstellen

Ereignisverbindungen:

- NmGetCfInfo.IND -> LOG_16.REQ
- LOG_16.CNF -> SD_A_1.REQ
- LOG_16.CNF -> SD_A_2.REQ
- LOG_16.CNF -> SD_A_3.REQ
- LOG_16.CNF -> SD_A_4.REQ

- LOG_16.CNF -> SD_A_5.REQ
- LOG_16.CNF -> SD_A_6.REQ
- LOG_16.CNF -> SD_A_7.REQ
- LOG_16.CNF -> SD_A_8.REQ
- LOG_16.CNF -> SD_A_9.REQ
- LOG_16.CNF -> SD_A_10.REQ
- LOG_16.CNF -> SD_A_11.REQ
- LOG_16.CNF -> SD_A_12.REQ
- LOG_16.CNF -> SD_A_13.REQ
- LOG_16.CNF -> SD_A_14.REQ
- LOG_16.CNF -> SD_A_15.REQ
- LOG_16.CNF -> SD_A_16.REQ
- SD_A_1.CNF -> NmSetNF_1.REQ
- NmSetNF_1.CNF -> SD_C_1.REQ
- SD_A_2.CNF -> NmSetNF_2.REQ
- NmSetNF_2.CNF -> SD_C_2.REQ
- SD_A_3.CNF -> NmSetNF_3.REQ
- NmSetNF_3.CNF -> SD_C_3.REQ
- SD_A_4.CNF -> NmSetNF_4.REQ
- NmSetNF_4.CNF -> SD_C_4.REQ
- SD_A_5.CNF -> NmSetNF_5.REQ
- NmSetNF_5.CNF -> SD_C_5.REQ
- SD_A_6.CNF -> NmSetNF_6.REQ
- NmSetNF_6.CNF -> SD_C_6.REQ
- SD_A_7.CNF -> NmSetNF_7.REQ
- NmSetNF_7.CNF -> SD_C_7.REQ
- SD_A_8.CNF -> NmSetNF_8.REQ
- NmSetNF_8.CNF -> SD_C_8.REQ
- SD_A_9.CNF -> NmSetNF_9.REQ
- NmSetNF_9.CNF -> SD_C_9.REQ
- SD_A_10.CNF -> NmSetNF_10.REQ
- NmSetNF_10.CNF -> SD_C_10.REQ
- SD_A_11.CNF -> NmSetNF_11.REQ
- NmSetNF_11.CNF -> SD_C_11.REQ
- SD_A_12.CNF -> NmSetNF_12.REQ
- NmSetNF_12.CNF -> SD_C_12.REQ
- SD_A_13.CNF -> NmSetNF_13.REQ
- NmSetNF_13.CNF -> SD_C_13.REQ
- SD_A_14.CNF -> NmSetNF_14.REQ
- NmSetNF_14.CNF -> SD_C_14.REQ
- SD_A_15.CNF -> NmSetNF_15.REQ
- NmSetNF_15.CNF -> SD_C_15.REQ
- SD_A_16.CNF -> NmSetNF_16.REQ
- NmSetNF_16.CNF -> SD_C_16.REQ
- LOG_16.CNF -> NmGetCfInfo.RSP

Datenverbindungen:

- NmGetCfInfo.sNetEv -> LOG_16.IN
- LOG_16.OUT1 -> SD_A_1.IN
- LOG_16.OUT2 -> SD_A_2.IN
- LOG_16.OUT3 -> SD_A_3.IN
- LOG_16.OUT4 -> SD_A_4.IN
- LOG_16.OUT5 -> SD_A_5.IN
- LOG_16.OUT6 -> SD_A_6.IN
- LOG_16.OUT7 -> SD_A_7.IN
- LOG_16.OUT8 -> SD_A_8.IN
- LOG_16.OUT9 -> SD_A_9.IN
- LOG_16.OUT10 -> SD_A_10.IN
- LOG_16.OUT11 -> SD_A_11.IN

- LOG_16.OUT12 -> SD_A_12.IN
- LOG_16.OUT13 -> SD_A_13.IN
- LOG_16.OUT14 -> SD_A_14.IN
- LOG_16.OUT15 -> SD_A_15.IN
- LOG_16.OUT16 -> SD_A_16.IN
- SD_A_1.cfName -> NmSetNF_1.au8IsoName
- SD_A_2.cfName -> NmSetNF_2.au8IsoName
- SD_A_3.cfName -> NmSetNF_3.au8IsoName
- SD_A_4.cfName -> NmSetNF_4.au8IsoName
- SD_A_5.cfName -> NmSetNF_5.au8IsoName
- SD_A_6.cfName -> NmSetNF_6.au8IsoName
- SD_A_7.cfName -> NmSetNF_7.au8IsoName
- SD_A_8.cfName -> NmSetNF_8.au8IsoName
- SD_A_9.cfName -> NmSetNF_9.au8IsoName
- SD_A_10.cfName -> NmSetNF_10.au8IsoName
- SD_A_11.cfName -> NmSetNF_11.au8IsoName
- SD_A_12.cfName -> NmSetNF_12.au8IsoName
- SD_A_13.cfName -> NmSetNF_13.au8IsoName
- SD_A_14.cfName -> NmSetNF_14.au8IsoName
- SD_A_15.cfName -> NmSetNF_15.au8IsoName
- SD_A_16.cfName -> NmSetNF_16.au8IsoName
- NmSetNF_1. -> SD_C_1.IN
- NmSetNF_2. -> SD_C_2.IN
- NmSetNF_3. -> SD_C_3.IN
- NmSetNF_4. -> SD_C_4.IN
- NmSetNF_5. -> SD_C_5.IN
- NmSetNF_6. -> SD_C_6.IN
- NmSetNF_7. -> SD_C_7.IN
- NmSetNF_8. -> SD_C_8.IN
- NmSetNF_9. -> SD_C_9.IN
- NmSetNF_10. -> SD_C_10.IN
- NmSetNF_11. -> SD_C_11.IN
- NmSetNF_12. -> SD_C_12.IN
- NmSetNF_13. -> SD_C_13.IN
- NmSetNF_14. -> SD_C_14.IN
- NmSetNF_15. -> SD_C_15.IN
- NmSetNF_16. -> SD_C_16.IN

Programmablauf und Funktionsweise

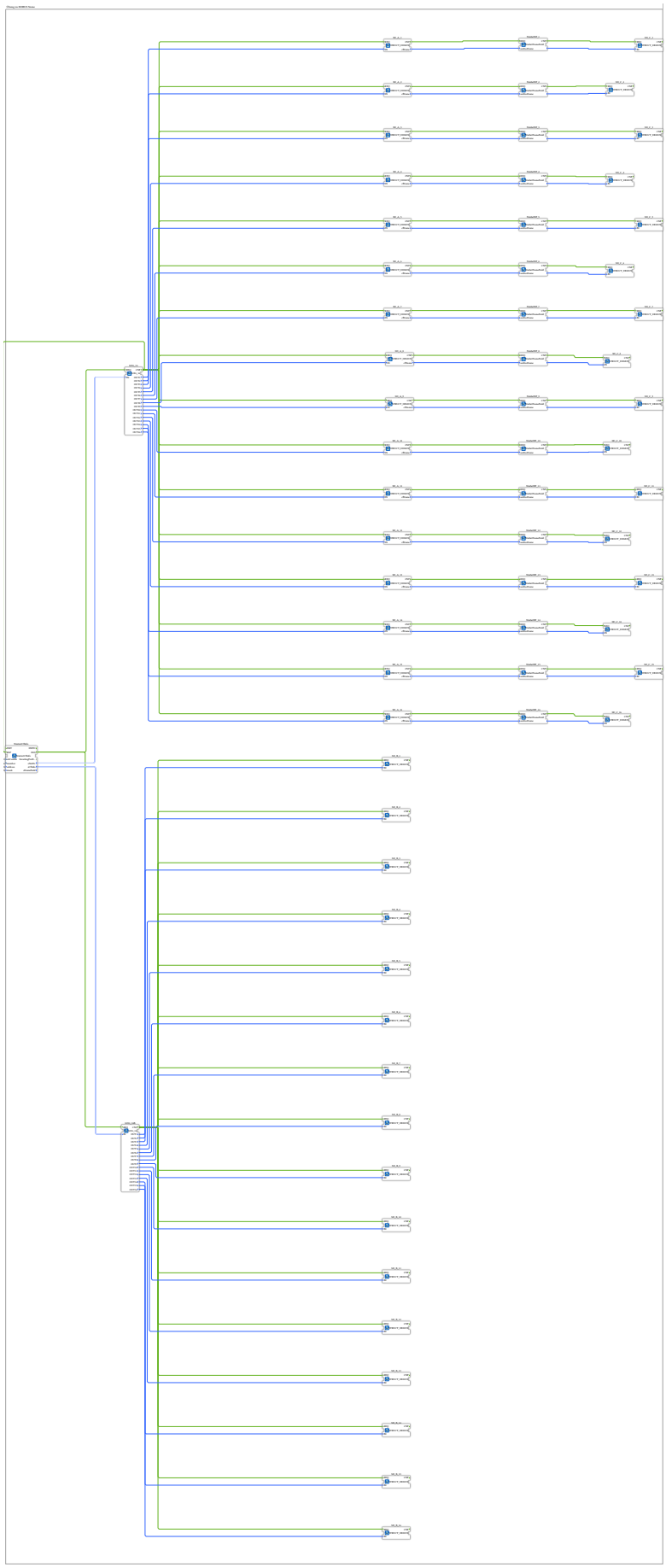
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_122_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.428 Uebung_122b_AX: Übung zu ISOBUS Name

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_122b_AX (Übung zu ISOBUS Name).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Übung zu ISOBUS Name**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_122b_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **NmGetCfInfo**: Instanz des Typs isobus::pgn::NmGetCfInfo.
  - Parameter u8CanIdx = NODE1
  - Parameter member = intern
  - Parameter address = ADD_ALL_PASS
  - Parameter mask = FLT_ALL_PASS
- **LOG_16**: Instanz des Typs logiBUS::utils::logging::LOG_16.
- **LOG_16B**: Instanz des Typs logiBUS::utils::logging::LOG_16.
- **SD_A_1**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_A_2**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_A_3**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_A_4**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_A_5**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_A_6**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_A_7**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_A_8**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_A_9**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_A_10**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_A_11**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_A_12**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_A_13**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_A_14**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_A_15**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_A_16**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_B_1**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_B_2**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_B_3**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_B_4**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_B_5**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_B_6**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_B_7**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_B_8**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_B_9**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_B_10**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_B_11**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_B_12**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_B_13**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_B_14**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_B_15**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_B_16**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **NmSetNF_1**: Instanz des Typs isobus::pgn::NmSetNameField.
- **NmSetNF_2**: Instanz des Typs isobus::pgn::NmSetNameField.
- **NmSetNF_3**: Instanz des Typs isobus::pgn::NmSetNameField.
- **NmSetNF_4**: Instanz des Typs isobus::pgn::NmSetNameField.
- **NmSetNF_5**: Instanz des Typs isobus::pgn::NmSetNameField.
- **NmSetNF_6**: Instanz des Typs isobus::pgn::NmSetNameField.
- **NmSetNF_7**: Instanz des Typs isobus::pgn::NmSetNameField.
- **NmSetNF_8**: Instanz des Typs isobus::pgn::NmSetNameField.
- **NmSetNF_9**: Instanz des Typs isobus::pgn::NmSetNameField.
- **NmSetNF_10**: Instanz des Typs isobus::pgn::NmSetNameField.
- **NmSetNF_11**: Instanz des Typs isobus::pgn::NmSetNameField.
- **NmSetNF_12**: Instanz des Typs isobus::pgn::NmSetNameField.

- **NmSetNF_13**: Instanz des Typs isobus::pgn::NmSetNameField.
- **NmSetNF_14**: Instanz des Typs isobus::pgn::NmSetNameField.
- **NmSetNF_15**: Instanz des Typs isobus::pgn::NmSetNameField.
- **NmSetNF_16**: Instanz des Typs isobus::pgn::NmSetNameField.
- **SD_C_1**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_C_2**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_C_3**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_C_4**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_C_5**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_C_6**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_C_7**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_C_8**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_C_9**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_C_10**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_C_11**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_C_12**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_C_13**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_C_14**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_C_15**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **SD_C_16**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.

Verbindungen und Schnittstellen

Ereignisverbindungen:

- NmGetCfInfo.IND -> LOG_16.REQ
- NmGetCfInfo.IND -> LOG_16B.REQ
- LOG_16.CNF -> SD_A_1.REQ
- LOG_16.CNF -> SD_A_2.REQ
- LOG_16.CNF -> SD_A_3.REQ
- LOG_16.CNF -> SD_A_4.REQ
- LOG_16.CNF -> SD_A_5.REQ
- LOG_16.CNF -> SD_A_6.REQ
- LOG_16.CNF -> SD_A_7.REQ
- LOG_16.CNF -> SD_A_8.REQ
- LOG_16.CNF -> SD_A_9.REQ
- LOG_16.CNF -> SD_A_10.REQ
- LOG_16.CNF -> SD_A_11.REQ
- LOG_16.CNF -> SD_A_12.REQ
- LOG_16.CNF -> SD_A_13.REQ
- LOG_16.CNF -> SD_A_14.REQ
- LOG_16.CNF -> SD_A_15.REQ
- LOG_16.CNF -> SD_A_16.REQ
- LOG_16B.CNF -> SD_B_1.REQ
- LOG_16B.CNF -> SD_B_2.REQ
- LOG_16B.CNF -> SD_B_3.REQ
- LOG_16B.CNF -> SD_B_4.REQ
- LOG_16B.CNF -> SD_B_5.REQ
- LOG_16B.CNF -> SD_B_6.REQ
- LOG_16B.CNF -> SD_B_7.REQ
- LOG_16B.CNF -> SD_B_8.REQ
- LOG_16B.CNF -> SD_B_9.REQ
- LOG_16B.CNF -> SD_B_10.REQ
- LOG_16B.CNF -> SD_B_11.REQ
- LOG_16B.CNF -> SD_B_12.REQ
- LOG_16B.CNF -> SD_B_13.REQ
- LOG_16B.CNF -> SD_B_14.REQ
- LOG_16B.CNF -> SD_B_15.REQ
- LOG_16B.CNF -> SD_B_16.REQ
- SD_A_1.CNF -> NmSetNF_1.REQ
- NmSetNF_1.CNF -> SD_C_1.REQ

- SD_A_2.CNF -> NmSetNF_2.REQ
- NmSetNF_2.CNF -> SD_C_2.REQ
- SD_A_3.CNF -> NmSetNF_3.REQ
- NmSetNF_3.CNF -> SD_C_3.REQ
- SD_A_4.CNF -> NmSetNF_4.REQ
- NmSetNF_4.CNF -> SD_C_4.REQ
- SD_A_5.CNF -> NmSetNF_5.REQ
- NmSetNF_5.CNF -> SD_C_5.REQ
- SD_A_6.CNF -> NmSetNF_6.REQ
- NmSetNF_6.CNF -> SD_C_6.REQ
- SD_A_7.CNF -> NmSetNF_7.REQ
- NmSetNF_7.CNF -> SD_C_7.REQ
- SD_A_8.CNF -> NmSetNF_8.REQ
- NmSetNF_8.CNF -> SD_C_8.REQ
- SD_A_9.CNF -> NmSetNF_9.REQ
- NmSetNF_9.CNF -> SD_C_9.REQ
- SD_A_10.CNF -> NmSetNF_10.REQ
- NmSetNF_10.CNF -> SD_C_10.REQ
- SD_A_11.CNF -> NmSetNF_11.REQ
- NmSetNF_11.CNF -> SD_C_11.REQ
- SD_A_12.CNF -> NmSetNF_12.REQ
- NmSetNF_12.CNF -> SD_C_12.REQ
- SD_A_13.CNF -> NmSetNF_13.REQ
- NmSetNF_13.CNF -> SD_C_13.REQ
- SD_A_14.CNF -> NmSetNF_14.REQ
- NmSetNF_14.CNF -> SD_C_14.REQ
- SD_A_15.CNF -> NmSetNF_15.REQ
- NmSetNF_15.CNF -> SD_C_15.REQ
- SD_A_16.CNF -> NmSetNF_16.REQ
- NmSetNF_16.CNF -> SD_C_16.REQ
- LOG_16.CNF -> NmGetCfInfo.RSP

Datenverbindungen:

- NmGetCfInfo.sNetEv -> LOG_16.IN
- NmGetCfInfo.sCfInfo -> LOG_16B.IN
- LOG_16.OUT1 -> SD_A_1.IN
- LOG_16.OUT2 -> SD_A_2.IN
- LOG_16.OUT3 -> SD_A_3.IN
- LOG_16.OUT4 -> SD_A_4.IN
- LOG_16.OUT5 -> SD_A_5.IN
- LOG_16.OUT6 -> SD_A_6.IN
- LOG_16.OUT7 -> SD_A_7.IN
- LOG_16.OUT8 -> SD_A_8.IN
- LOG_16.OUT9 -> SD_A_9.IN
- LOG_16.OUT10 -> SD_A_10.IN
- LOG_16.OUT11 -> SD_A_11.IN
- LOG_16.OUT12 -> SD_A_12.IN
- LOG_16.OUT13 -> SD_A_13.IN
- LOG_16.OUT14 -> SD_A_14.IN
- LOG_16.OUT15 -> SD_A_15.IN
- LOG_16.OUT16 -> SD_A_16.IN
- LOG_16B.OUT1 -> SD_B_1.IN
- LOG_16B.OUT2 -> SD_B_2.IN
- LOG_16B.OUT3 -> SD_B_3.IN
- LOG_16B.OUT4 -> SD_B_4.IN
- LOG_16B.OUT5 -> SD_B_5.IN
- LOG_16B.OUT6 -> SD_B_6.IN
- LOG_16B.OUT7 -> SD_B_7.IN
- LOG_16B.OUT8 -> SD_B_8.IN

- LOG_16B.OUT9 -> SD_B_9.IN
- LOG_16B.OUT10 -> SD_B_10.IN
- LOG_16B.OUT11 -> SD_B_11.IN
- LOG_16B.OUT12 -> SD_B_12.IN
- LOG_16B.OUT13 -> SD_B_13.IN
- LOG_16B.OUT14 -> SD_B_14.IN
- LOG_16B.OUT15 -> SD_B_15.IN
- LOG_16B.OUT16 -> SD_B_16.IN
- SD_A_1.cfName -> NmSetNF_1.au8IsoName
- SD_A_2.cfName -> NmSetNF_2.au8IsoName
- SD_A_3.cfName -> NmSetNF_3.au8IsoName
- SD_A_4.cfName -> NmSetNF_4.au8IsoName
- SD_A_5.cfName -> NmSetNF_5.au8IsoName
- SD_A_6.cfName -> NmSetNF_6.au8IsoName
- SD_A_7.cfName -> NmSetNF_7.au8IsoName
- SD_A_8.cfName -> NmSetNF_8.au8IsoName
- SD_A_9.cfName -> NmSetNF_9.au8IsoName
- SD_A_10.cfName -> NmSetNF_10.au8IsoName
- SD_A_11.cfName -> NmSetNF_11.au8IsoName
- SD_A_12.cfName -> NmSetNF_12.au8IsoName
- SD_A_13.cfName -> NmSetNF_13.au8IsoName
- SD_A_14.cfName -> NmSetNF_14.au8IsoName
- SD_A_15.cfName -> NmSetNF_15.au8IsoName
- SD_A_16.cfName -> NmSetNF_16.au8IsoName
- NmSetNF_1. -> SD_C_1.IN
- NmSetNF_2. -> SD_C_2.IN
- NmSetNF_3. -> SD_C_3.IN
- NmSetNF_4. -> SD_C_4.IN
- NmSetNF_5. -> SD_C_5.IN
- NmSetNF_6. -> SD_C_6.IN
- NmSetNF_7. -> SD_C_7.IN
- NmSetNF_8. -> SD_C_8.IN
- NmSetNF_9. -> SD_C_9.IN
- NmSetNF_10. -> SD_C_10.IN
- NmSetNF_11. -> SD_C_11.IN
- NmSetNF_12. -> SD_C_12.IN
- NmSetNF_13. -> SD_C_13.IN
- NmSetNF_14. -> SD_C_14.IN
- NmSetNF_15. -> SD_C_15.IN
- NmSetNF_16. -> SD_C_16.IN

Programmablauf und Funktionsweise

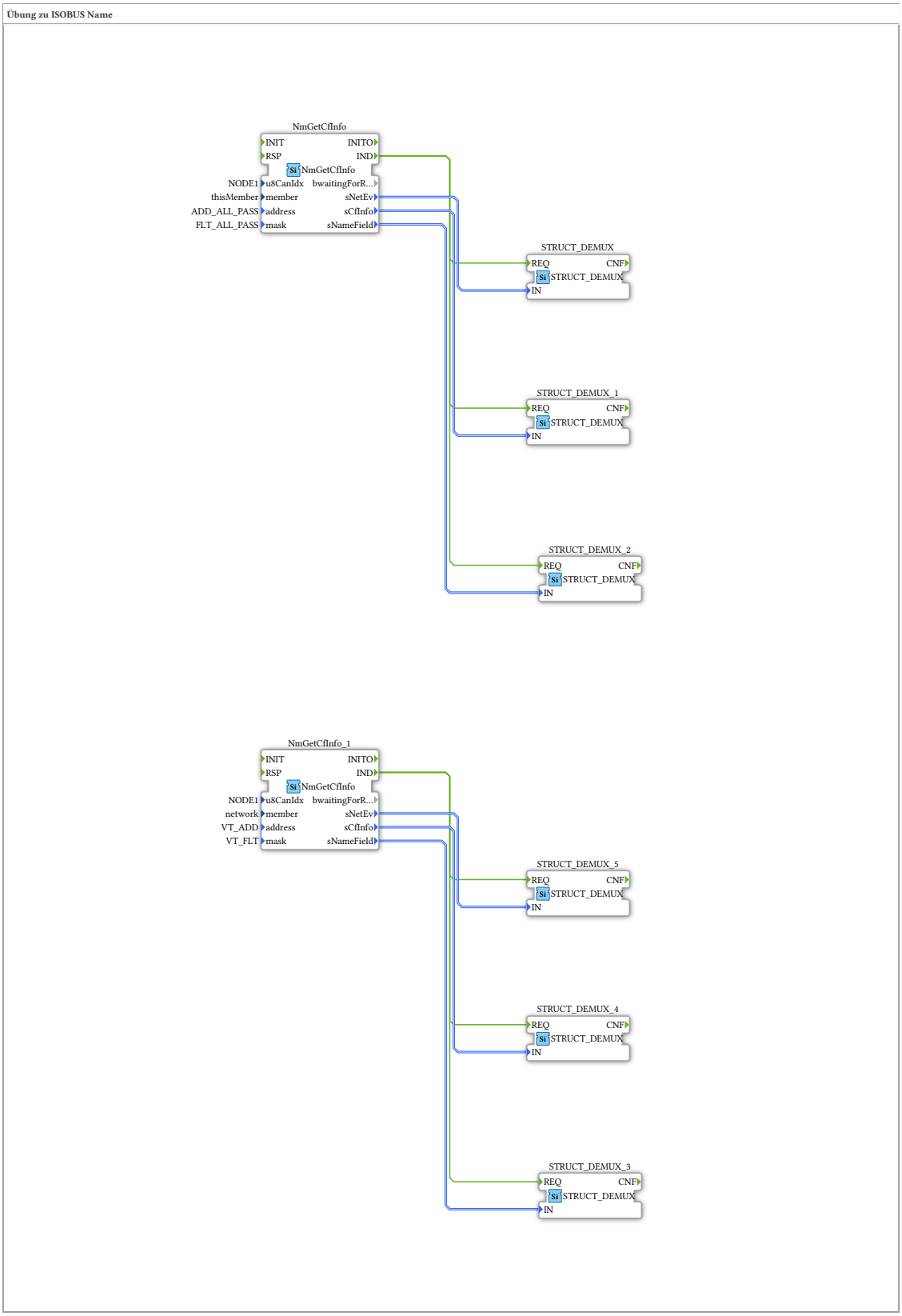
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_122b_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.429 Uebung_123_AX: Übung zu ISOBUS Name

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_123_AX (Übung zu ISOBUS Name).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Übung zu ISOBUS Name**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_123_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **NmGetCfInfo**: Instanz des Typs isobus::pgn::NmGetCfInfo.
  - Parameter u8CanIdx = NODE1
  - Parameter member = thisMember
  - Parameter address = ADD_ALL_PASS
  - Parameter mask = FLT_ALL_PASS
- **STRUCT_DEMUX**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **STRUCT_DEMUX_1**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **STRUCT_DEMUX_2**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **STRUCT_DEMUX_3**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **NmGetCfInfo_1**: Instanz des Typs isobus::pgn::NmGetCfInfo.
  - Parameter u8CanIdx = NODE1
  - Parameter member = network
  - Parameter address = VT_ADD
  - Parameter mask = VT_FLT
- **STRUCT_DEMUX_4**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **STRUCT_DEMUX_5**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.

Verbindungen und Schnittstellen

Ereignisverbindungen:

- NmGetCfInfo.IND -> STRUCT_DEMUX.REQ
- NmGetCfInfo.IND -> STRUCT_DEMUX_1.REQ
- NmGetCfInfo.IND -> STRUCT_DEMUX_2.REQ
- NmGetCfInfo_1.IND -> STRUCT_DEMUX_3.REQ
- NmGetCfInfo_1.IND -> STRUCT_DEMUX_4.REQ
- NmGetCfInfo_1.IND -> STRUCT_DEMUX_5.REQ

Datenverbindungen:

- NmGetCfInfo.sNetEv -> STRUCT_DEMUX.IN
- NmGetCfInfo.sCfInfo -> STRUCT_DEMUX_1.IN
- NmGetCfInfo.sNameField -> STRUCT_DEMUX_2.IN
- NmGetCfInfo_1.sNameField -> STRUCT_DEMUX_3.IN
- NmGetCfInfo_1.sNetEv -> STRUCT_DEMUX_5.IN
- NmGetCfInfo_1.sCfInfo -> STRUCT_DEMUX_4.IN

Programmablauf und Funktionsweise

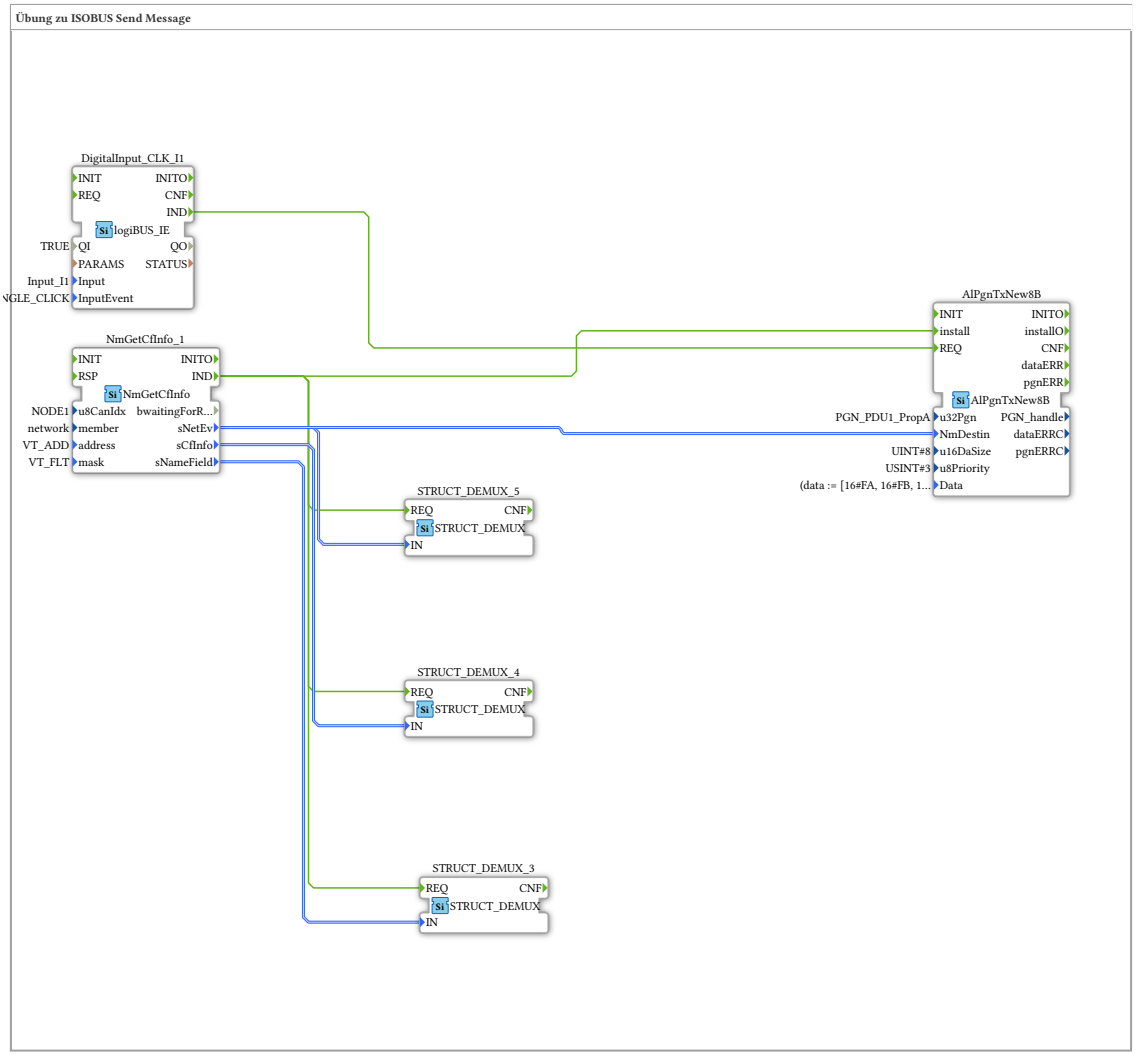
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_123_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.430 Uebung_124_AX: Übung zu ISOBUS Send Message

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_124_AX (Übung zu ISOBUS Send Message).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Übung zu ISOBUS Send Message**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_124_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **STRUCT_DEMUX_3**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **NmGetCfInfo_1**: Instanz des Typs isobus::pgn::NmGetCfInfo.
  - Parameter u8CanIdx = NODE1
  - Parameter member = network
  - Parameter address = VT_ADD
  - Parameter mask = VT_FLT
- **STRUCT_DEMUX_4**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **STRUCT_DEMUX_5**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **AlPgnTxNew8B**: Instanz des Typs isobus::pgn::tx::AlPgnTxNew8B.
  - Parameter u32Pgn = PGN_PDU1_PropA
  - Parameter u16DaSize = 8
  - Parameter u8Priority = 3
  - Parameter Data = (data := [16#FA, 16#FB, 16#FC, 16#FD, 16#FE, 16#FF, 16#F1, 16#F2])
- **DigitalInput_CLK_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE

- Parameter Input = Input_I1
- Parameter InputEvent = BUTTON_SINGLE_CLICK

Verbindungen und Schnittstellen

Ereignisverbindungen:

- NmGetCfInfo_1.IND -> STRUCT_DEMUX_3.REQ
- NmGetCfInfo_1.IND -> STRUCT_DEMUX_4.REQ
- NmGetCfInfo_1.IND -> STRUCT_DEMUX_5.REQ
- NmGetCfInfo_1.IND -> AlPgnTxNew8B.install
- DigitalInput_CLK_I1.IND -> AlPgnTxNew8B.REQ

Datenverbindungen:

- NmGetCfInfo_1.sNameField -> STRUCT_DEMUX_3.IN
- NmGetCfInfo_1.sNetEv -> STRUCT_DEMUX_5.IN
- NmGetCfInfo_1.sCfInfo -> STRUCT_DEMUX_4.IN
- NmGetCfInfo_1.sNetEv -> AlPgnTxNew8B.NmDestin

Programmablauf und Funktionsweise

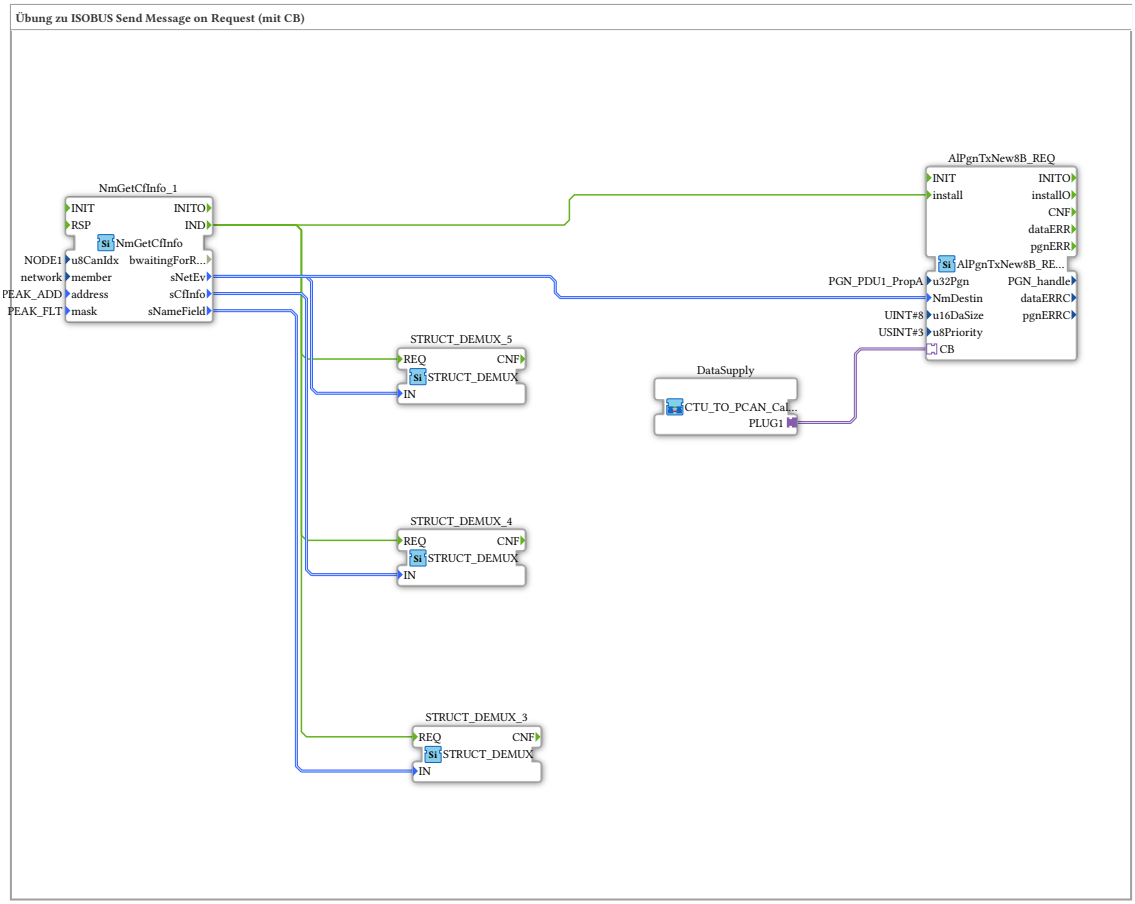
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_124_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.431 Uebung_125_AX: Übung zu ISOBUS Send Message on Request (mit CB)

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_125_AX (Übung zu ISOBUS Send Message on Request (mit CB)).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Übung zu ISOBUS Send Message on Request (mit CB)**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_125_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **STRUCT_DEMUX_3**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **NmGetCfInfo_1**: Instanz des Typs isobus::pgn::NmGetCfInfo.
  - Parameter u8CanIdx = NODE1
  - Parameter member = network
  - Parameter address = PEAK_ADD
  - Parameter mask = PEAK_FLT
- **STRUCT_DEMUX_4**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **STRUCT_DEMUX_5**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **AlPgnTxNew8B_REQ**: Instanz des Typs isobus::pgn::tx::AlPgnTxNew8B_REQ.
  - Parameter u32Pgn = PGN_PDU1_PropA
  - Parameter u16DaSize = 8
  - Parameter u8Priority = 3

Verbindungen und Schnittstellen

Adapterverbindungen:

- DataSupply.PLUG1 -> AlPgnTxNew8B_REQ.CB

Ereignisverbindungen:

- NmGetCfInfo_1.IND -> STRUCT_DEMUX_3.REQ

- NmGetCfInfo_1.IND -> STRUCT_DEMUX_4.REQ
- NmGetCfInfo_1.IND -> STRUCT_DEMUX_5.REQ
- NmGetCfInfo_1.IND -> AlPgnTxNew8B_REQ.install

Datenverbindungen:

- NmGetCfInfo_1.sNameField -> STRUCT_DEMUX_3.IN
- NmGetCfInfo_1.sNetEv -> STRUCT_DEMUX_5.IN
- NmGetCfInfo_1.sCfInfo -> STRUCT_DEMUX_4.IN
- NmGetCfInfo_1.sNetEv -> AlPgnTxNew8B_REQ.NmDestin

Programmablauf und Funktionsweise

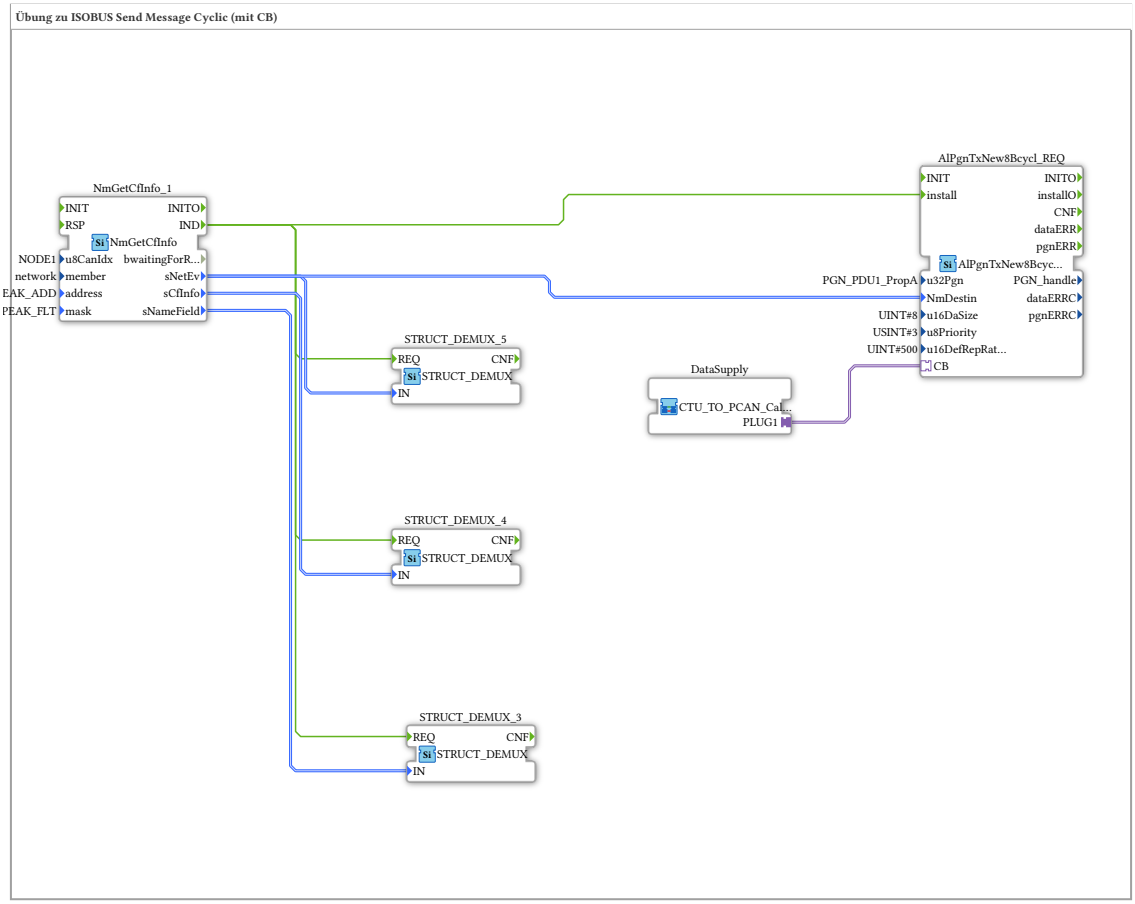
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_125_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.432 Uebung_126_AX: Übung zu ISOBUS Send Message Cyclic (mit CB)

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_126_AX (Übung zu ISOBUS Send Message Cyclic (mit CB)).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Übung zu ISOBUS Send Message Cyclic (mit CB)**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_126_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **STRUCT_DEMUX_3**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **NmGetCfInfo_1**: Instanz des Typs isobus::pgn::NmGetCfInfo.
  - Parameter u8CanIdx = NODE1
  - Parameter member = network
  - Parameter address = PEAK_ADD
  - Parameter mask = PEAK_FLT
- **STRUCT_DEMUX_4**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **STRUCT_DEMUX_5**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **AlPgnTxNew8Bcycl_REQ**: Instanz des Typs isobus::pgn::tx::AlPgnTxNew8Bcycl_REQ.
  - Parameter u32Pgn = PGN_PDU1_PropA
  - Parameter u16DaSize = 8
  - Parameter u8Priority = 3
  - Parameter u16DefRepRate = 500

Verbindungen und Schnittstellen

Adapterverbindungen:

- DataSupply.PLUG1 -> AlPgnTxNew8Bcycl_REQ.CB

Ereignisverbindungen:

- NmGetCfInfo_1.IND -> STRUCT_DEMUX_3.REQ
- NmGetCfInfo_1.IND -> STRUCT_DEMUX_4.REQ
- NmGetCfInfo_1.IND -> STRUCT_DEMUX_5.REQ
- NmGetCfInfo_1.IND -> AlPgnTxNew8Bcycl_REQ.install

Datenverbindungen:

- NmGetCfInfo_1.sNameField -> STRUCT_DEMUX_3.IN
- NmGetCfInfo_1.sNetEv -> STRUCT_DEMUX_5.IN
- NmGetCfInfo_1.sCfInfo -> STRUCT_DEMUX_4.IN
- NmGetCfInfo_1.sNetEv -> AlPgnTxNew8Bcycl_REQ.NmDestin

Programmablauf und Funktionsweise

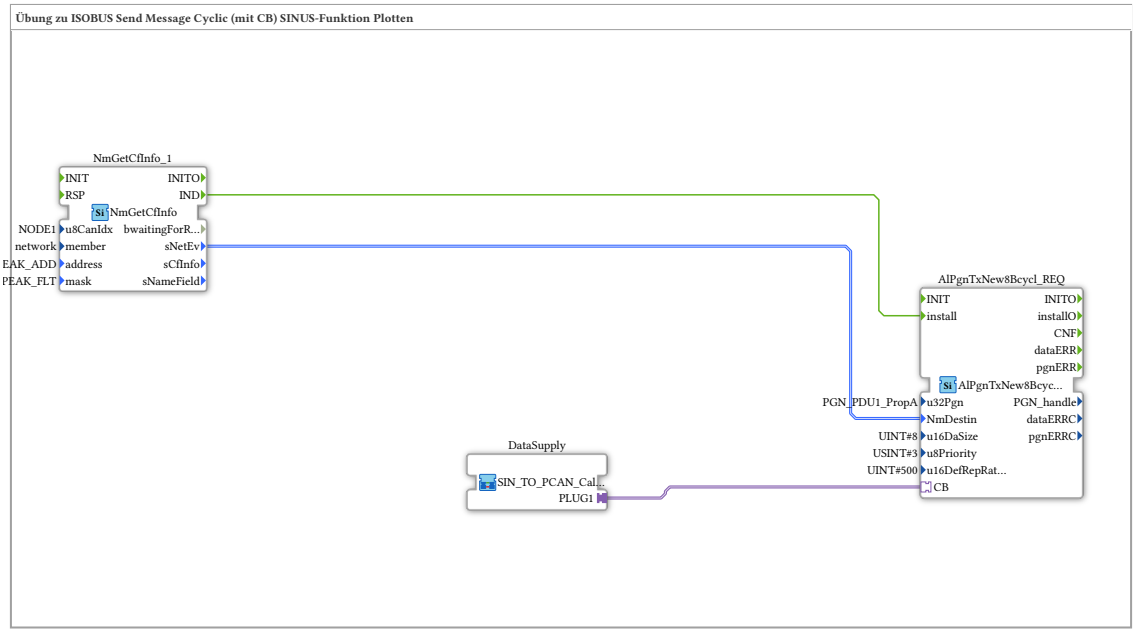
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_126_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.433 Uebung_126b2_AX: Übung zu ISOBUS Send Message Cyclic (mit CB) SINUS-Funktion Plotten

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_126b2_AX (Übung zu ISOBUS Send Message Cyclic (mit CB) SINUS-Funktion Plotten).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Übung zu ISOBUS Send Message Cyclic (mit CB) SINUS-Funktion Plotten**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_126b2_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **NmGetCfInfo_1**: Instanz des Typs isobus::pgn::NmGetCfInfo.
  - Parameter u8CanIdx = NODE1
  - Parameter member = network
  - Parameter address = PEAK_ADD
  - Parameter mask = PEAK_FLT
- **AlPgnTxNew8Bcycl_REQ**: Instanz des Typs isobus::pgn::tx::AlPgnTxNew8Bcycl_REQ.
  - Parameter u32Pgn = PGN_PDU1_PropA
  - Parameter u16DaSize = 8
  - Parameter u8Priority = 3
  - Parameter u16DefRepRate = 500

Verbindungen und Schnittstellen

Adapterverbindungen:

- DataSupply.PLUG1 -> AlPgnTxNew8Bcycl_REQ.CB

Ereignisverbindungen:

- NmGetCfInfo_1.IND -> AlPgnTxNew8Bcycl_REQ.install

Datenverbindungen:

- NmGetCfInfo_1.sNetEv -> AlPgnTxNew8Bcycl_REQ.NmDestin

Programmablauf und Funktionsweise

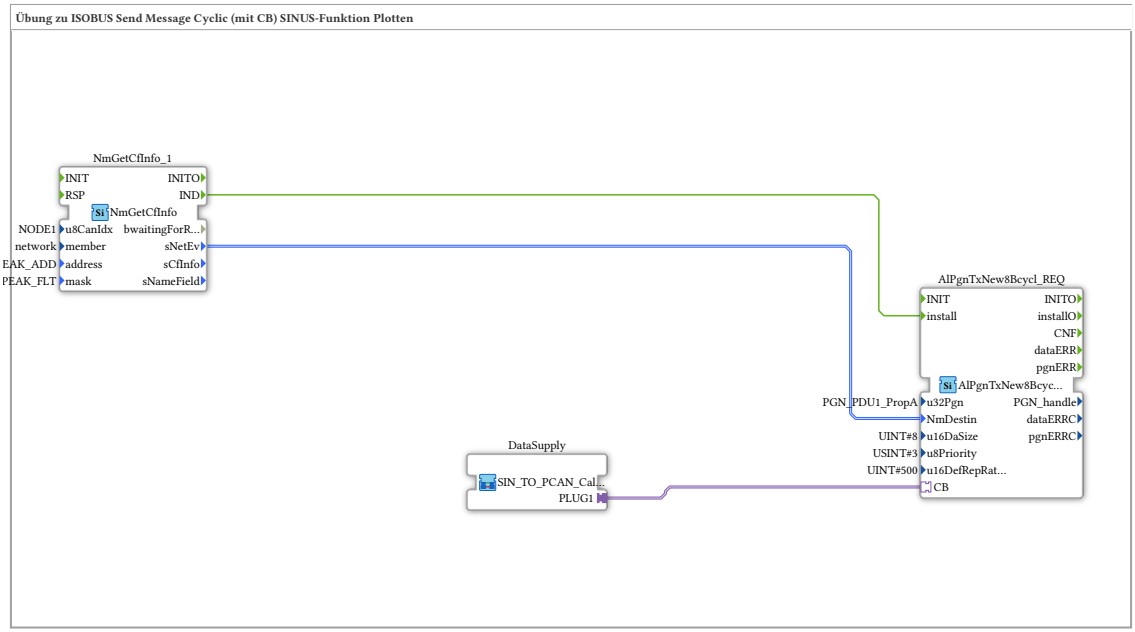
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_126b2_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.434 Uebung_126b_AX: Übung zu ISOBUS Send Message Cyclic (mit CB) SINUS-Funktion Plotten

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_126b_AX (Übung zu ISOBUS Send Message Cyclic (mit CB) SINUS-Funktion Plotten).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Übung zu ISOBUS Send Message Cyclic (mit CB) SINUS-Funktion Plotten**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_126b_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **NmGetCfInfo_1**: Instanz des Typs isobus::pgn::NmGetCfInfo.
  - Parameter u8CanIdx = NODE1
  - Parameter member = network
  - Parameter address = PEAK_ADD
  - Parameter mask = PEAK_FLT
- **AlPgnTxNew8Bcycl_REQ**: Instanz des Typs isobus::pgn::tx::AlPgnTxNew8Bcycl_REQ.
  - Parameter u32Pgn = PGN_PDU1_PropA
  - Parameter u16DaSize = 8
  - Parameter u8Priority = 3
  - Parameter u16DefRepRate = 500

Verbindungen und Schnittstellen

Adapterverbindungen:

- DataSupply.PLUG1 -> AlPgnTxNew8Bcycl_REQ.CB

Ereignisverbindungen:

- NmGetCfInfo_1.IND -> AlPgnTxNew8Bcycl_REQ.install

Datenverbindungen:

- NmGetCfInfo_1.sNetEv -> AlPgnTxNew8Bcycl_REQ.NmDestin

Programmablauf und Funktionsweise

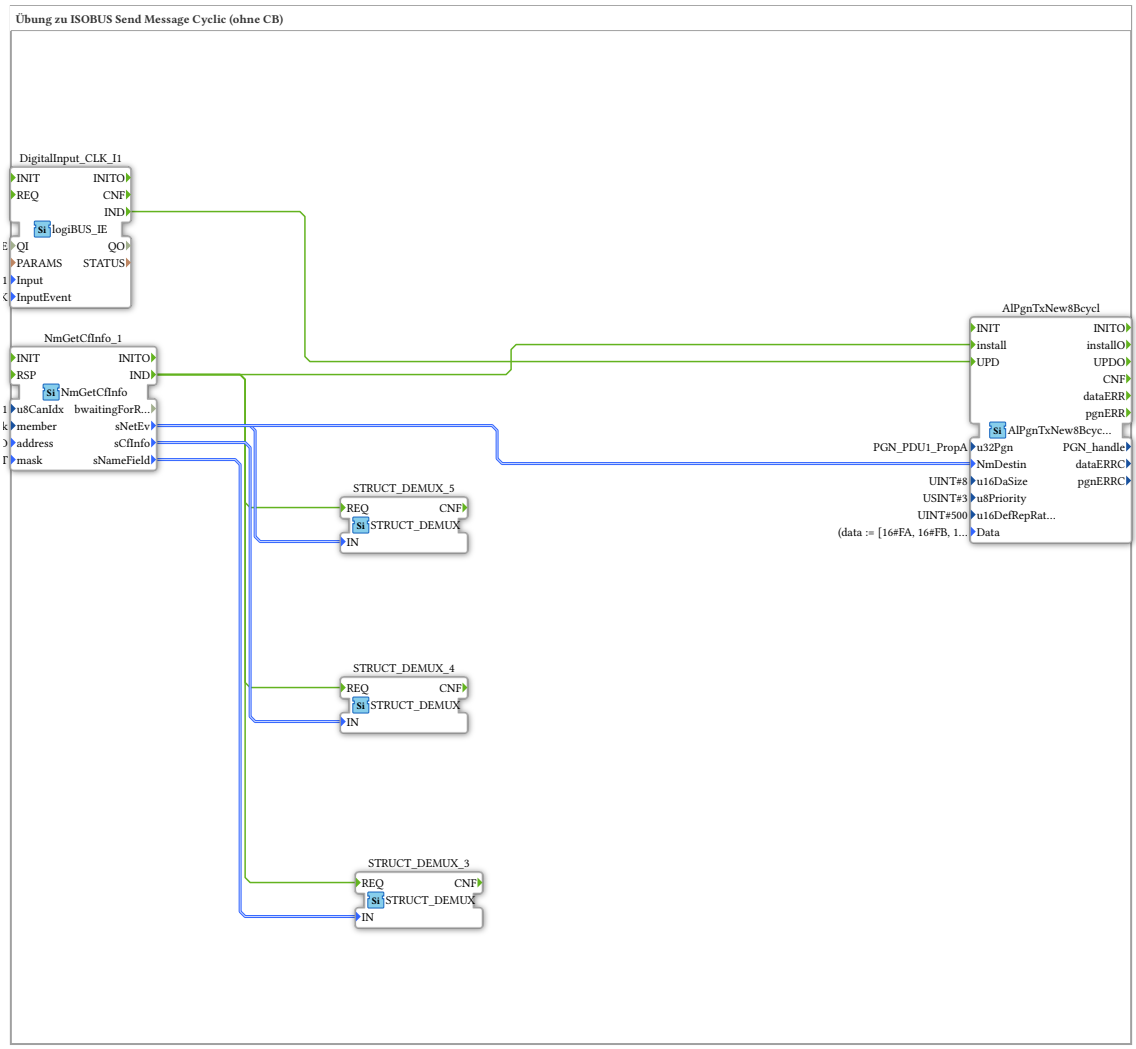
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_126b_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.435 Uebung_127_AX: Übung zu ISOBUS Send Message Cyclic (ohne CB)

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_127_AX (Übung zu ISOBUS Send Message Cyclic (ohne CB)).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Übung zu ISOBUS Send Message Cyclic (ohne CB)**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_127_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **STRUCT_DEMUX_3**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **NmGetCfInfo_1**: Instanz des Typs isobus::pgn::NmGetCfInfo.
  - Parameter u8CanIdx = NODE1
  - Parameter member = network
  - Parameter address = VT_ADD
  - Parameter mask = VT_FLT
- **STRUCT_DEMUX_4**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **STRUCT_DEMUX_5**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **AlPgnTxNew8Bcycl**: Instanz des Typs isobus::pgn::tx::AlPgnTxNew8Bcycl.
  - Parameter u32Pgn = PGN_PDU1_PropA
  - Parameter u16DaSize = 8
  - Parameter u8Priority = 3
  - Parameter u16DefRepRate = 500
  - Parameter Data = (data := [16#FA, 16#FB, 16#FC, 16#FD, 16#FE, 16#FF, 16#F1, 16#F2])
- **DigitalInput_CLK_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
  - Parameter InputEvent = BUTTON_SINGLE_CLICK

Verbindungen und Schnittstellen

Ereignisverbindungen:

- NmGetCfInfo_1.IND -> STRUCT_DEMUX_3.REQ
- NmGetCfInfo_1.IND -> STRUCT_DEMUX_4.REQ
- NmGetCfInfo_1.IND -> STRUCT_DEMUX_5.REQ
- NmGetCfInfo_1.IND -> AlPgnTxNew8Bcycl.install
- DigitalInput_CLK_I1.IND -> AlPgnTxNew8Bcycl.UPD

Datenverbindungen:

- NmGetCfInfo_1.sNameField -> STRUCT_DEMUX_3.IN
- NmGetCfInfo_1.sNetEv -> STRUCT_DEMUX_5.IN
- NmGetCfInfo_1.sCfInfo -> STRUCT_DEMUX_4.IN
- NmGetCfInfo_1.sNetEv -> AlPgnTxNew8Bcycl.NmDestin

Programmablauf und Funktionsweise

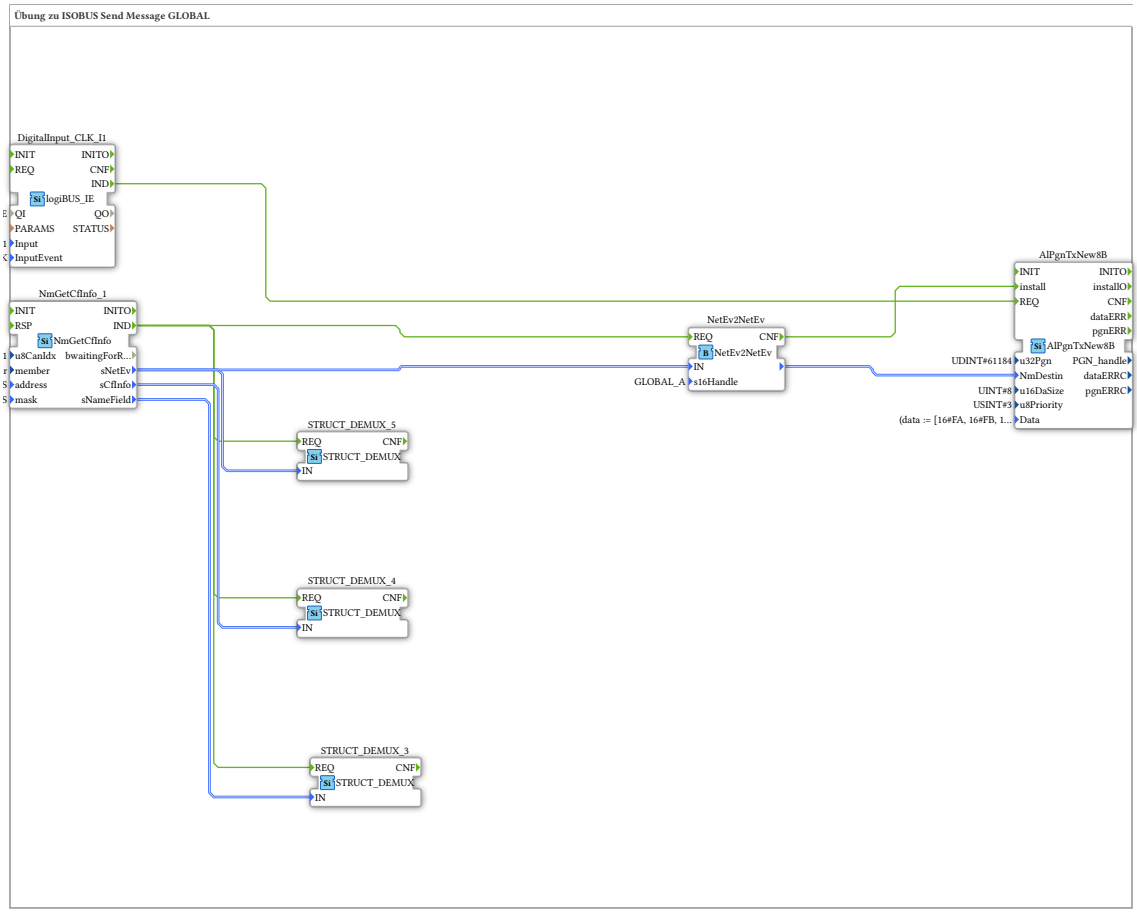
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_127_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.436 Uebung_128_AX: Übung zu ISOBUS Send Message GLOBAL

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_128_AX (Übung zu ISOBUS Send Message GLOBAL).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Übung zu ISOBUS Send Message GLOBAL**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_128_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **STRUCT_DEMUX_3**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **NmGetCfInfo_1**: Instanz des Typs isobus::pgn::NmGetCfInfo.
  - Parameter u8CanIdx = NODE1
  - Parameter member = thisMember
  - Parameter address = ADD_ALL_PASS
  - Parameter mask = FLT_ALL_PASS
- **STRUCT_DEMUX_4**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **STRUCT_DEMUX_5**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **AlPgnTxNew8B**: Instanz des Typs isobus::pgn::tx::AlPgnTxNew8B.
  - Parameter u32Pgn = 61184
  - Parameter u16DaSize = 8
  - Parameter u8Priority = 3
  - Parameter Data = (data := [16#FA, 16#FB, 16#FC, 16#FD, 16#FE, 16#FF, 16#F1, 16#F2])
- **DigitalInput_CLK_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **NetEv2NetEv**: Instanz des Typs isobus::pgn::NetEv2NetEv.
  - Parameter s16Handle = GLOBAL_A

Verbindungen und Schnittstellen

Ereignisverbindungen:

- NmGetCfInfo_1.IND -> STRUCT_DEMUX_3.REQ
- NmGetCfInfo_1.IND -> STRUCT_DEMUX_4.REQ
- NmGetCfInfo_1.IND -> STRUCT_DEMUX_5.REQ
- NmGetCfInfo_1.IND -> NetEv2NetEv.REQ
- DigitalInput_CLK_I1.IND -> AlPgnTxNew8B.REQ
- NetEv2NetEv.CNF -> AlPgnTxNew8B.install

Datenverbindungen:

- NmGetCfInfo_1.sNameField -> STRUCT_DEMUX_3.IN
- NmGetCfInfo_1.sNetEv -> STRUCT_DEMUX_5.IN
- NmGetCfInfo_1.sCfInfo -> STRUCT_DEMUX_4.IN
- NmGetCfInfo_1.sNetEv -> NetEv2NetEv.IN
- NetEv2NetEv. -> AlPgnTxNew8B.NmDestin

Programmablauf und Funktionsweise

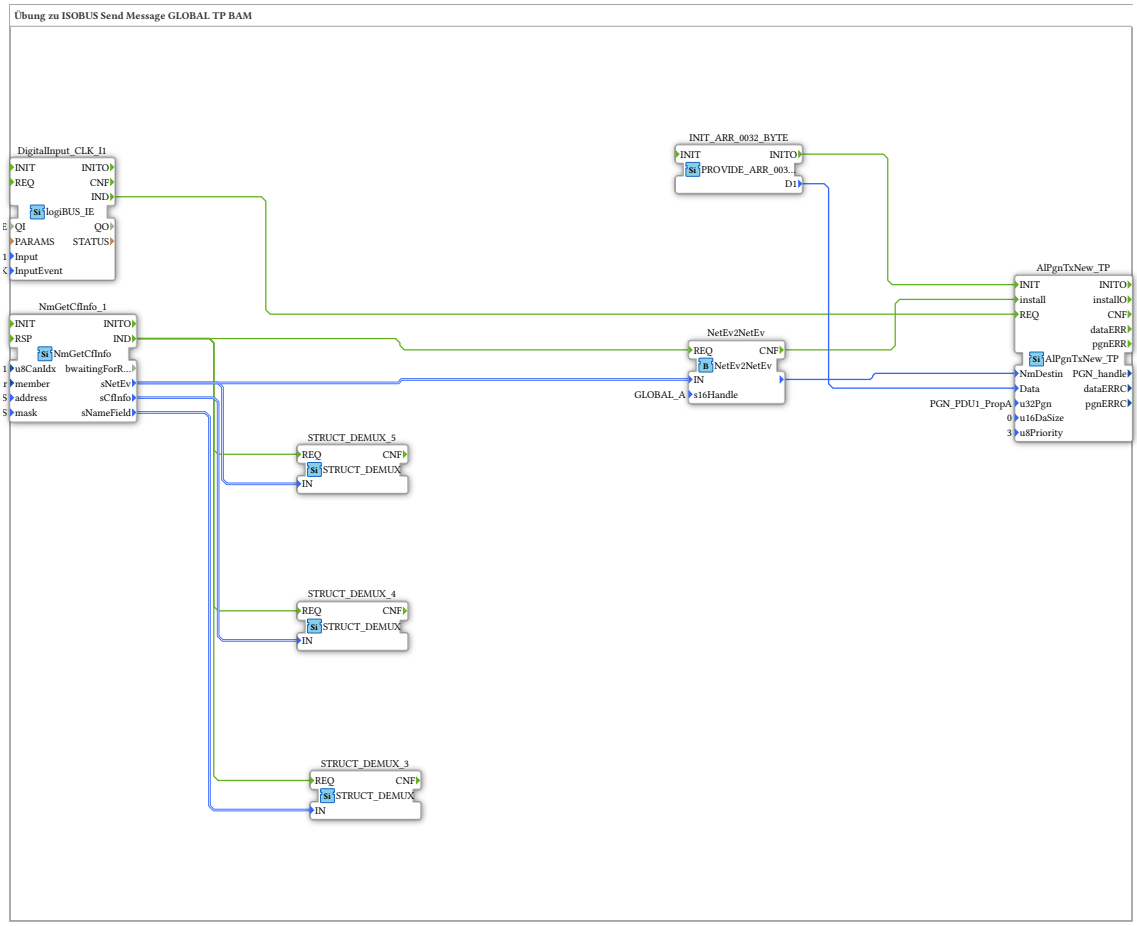
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_128_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.437 Uebung_128b_AX: Übung zu ISOBUS Send Message GLOBAL TP BAM

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_128b_AX (Übung zu ISOBUS Send Message GLOBAL TP BAM).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Übung zu ISOBUS Send Message GLOBAL TP BAM**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_128b_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **STRUCT_DEMUX_3**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **NmGetCfInfo_1**: Instanz des Typs isobus::pgn::NmGetCfInfo.
  - Parameter u8CanIdx = NODE1
  - Parameter member = thisMember
  - Parameter address = ADD_ALL_PASS
  - Parameter mask = FLT_ALL_PASS
- **STRUCT_DEMUX_4**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **STRUCT_DEMUX_5**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **AlPgnTxNew_TP**: Instanz des Typs isobus::pgn::tx::AlPgnTxNew_TP.
  - Parameter u32Pgn = PGN_PDU1_PropA
  - Parameter u16DaSize = 0
  - Parameter u8Priority = 3
- **DigitalInput_CLK_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **NetEv2NetEv**: Instanz des Typs isobus::pgn::NetEv2NetEv.
  - Parameter s16Handle = GLOBAL_A

- **INIT_ARR_0032_BYTE**: Instanz des Typs eclipse4diac::convert::providers::PROVIDE_ARR_0032_BYTE.
  - Parameter D1 = [16#00, 16#00]

Verbindungen und Schnittstellen

Ereignisverbindungen:

- NmGetCfInfo_1.IND -> STRUCT_DEMUX_3.REQ
- NmGetCfInfo_1.IND -> STRUCT_DEMUX_4.REQ
- NmGetCfInfo_1.IND -> STRUCT_DEMUX_5.REQ
- NmGetCfInfo_1.IND -> NetEv2NetEv.REQ
- DigitalInput_CLK_I1.IND -> AlPgnTxNew_TP.REQ
- NetEv2NetEv.CNF -> AlPgnTxNew_TP.install
- INIT_ARR_0032_BYTE.INITO -> AlPgnTxNew_TP.INIT

Datenverbindungen:

- NmGetCfInfo_1.sNameField -> STRUCT_DEMUX_3.IN
- NmGetCfInfo_1.sNetEv -> STRUCT_DEMUX_5.IN
- NmGetCfInfo_1.sCfInfo -> STRUCT_DEMUX_4.IN
- NmGetCfInfo_1.sNetEv -> NetEv2NetEv.IN
- NetEv2NetEv. -> AlPgnTxNew_TP.NmDestin
- INIT_ARR_0032_BYTE.D1 -> AlPgnTxNew_TP.Data

Programmablauf und Funktionsweise

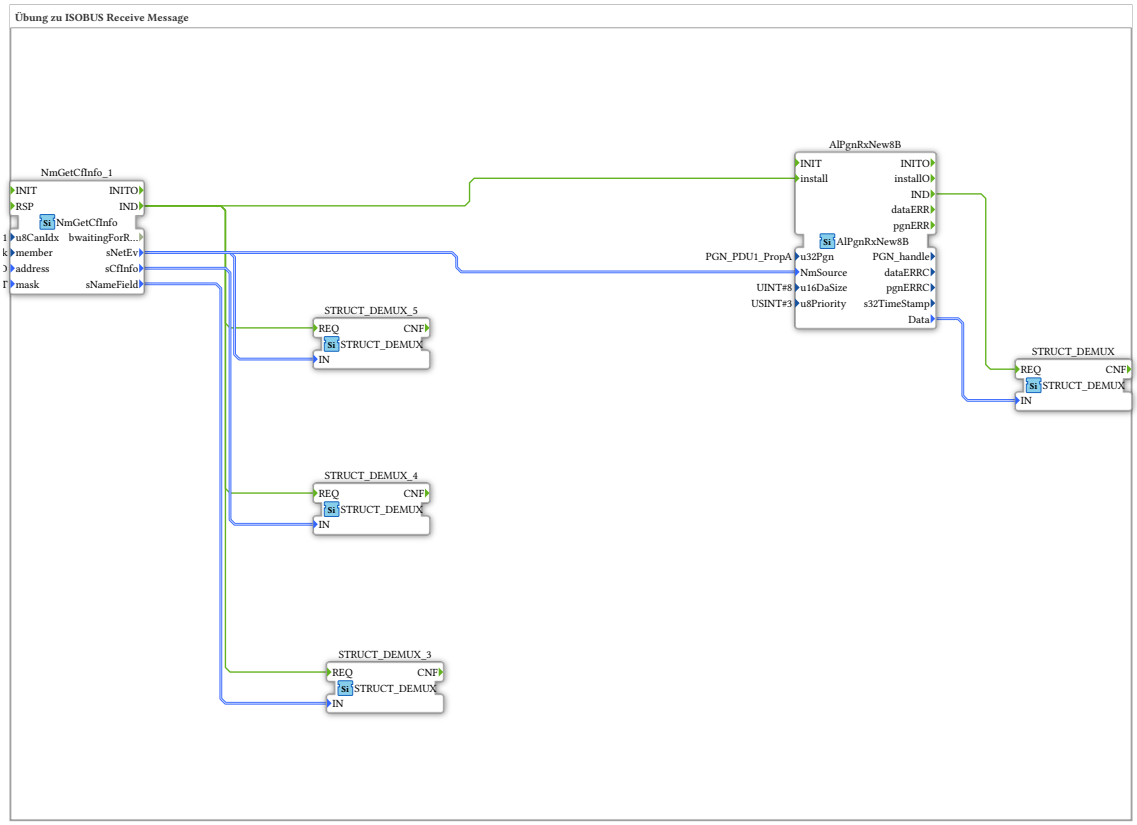
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_128b_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.438 Uebung_130_AX: Übung zu ISOBUS Receive Message

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_130_AX (Übung zu ISOBUS Receive Message).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Übung zu ISOBUS Receive Message**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_130_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **STRUCT_DEMUX_3**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **NmGetCfInfo_1**: Instanz des Typs isobus::pgn::NmGetCfInfo.
  - Parameter u8CanIdx = NODE1
  - Parameter member = network
  - Parameter address = PEAK_ADD
  - Parameter mask = PEAK_FLT
- **STRUCT_DEMUX_4**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **STRUCT_DEMUX_5**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **AlPgnRxNew8B**: Instanz des Typs isobus::pgn::rx::AlPgnRxNew8B.
  - Parameter u32Pgn = PGN_PDU1_PropA
  - Parameter u16DaSize = 8
  - Parameter u8Priority = 3
- **STRUCT_DEMUX**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.

Verbindungen und Schnittstellen

Ereignisverbindungen:

- NmGetCfInfo_1.IND -> STRUCT_DEMUX_3.REQ
- NmGetCfInfo_1.IND -> STRUCT_DEMUX_4.REQ
- NmGetCfInfo_1.IND -> STRUCT_DEMUX_5.REQ
- NmGetCfInfo_1.IND -> AlPgnRxNew8B.install
- AlPgnRxNew8B.IND -> STRUCT_DEMUX.REQ

Datenverbindungen:

- NmGetCfInfo_1.sNameField -> STRUCT_DEMUX_3.IN
- NmGetCfInfo_1.sNetEv -> STRUCT_DEMUX_5.IN
- NmGetCfInfo_1.sCfInfo -> STRUCT_DEMUX_4.IN
- NmGetCfInfo_1.sNetEv -> AlPgnRxNew8B.NmSource
- AlPgnRxNew8B.Data -> STRUCT_DEMUX.IN

Programmablauf und Funktionsweise

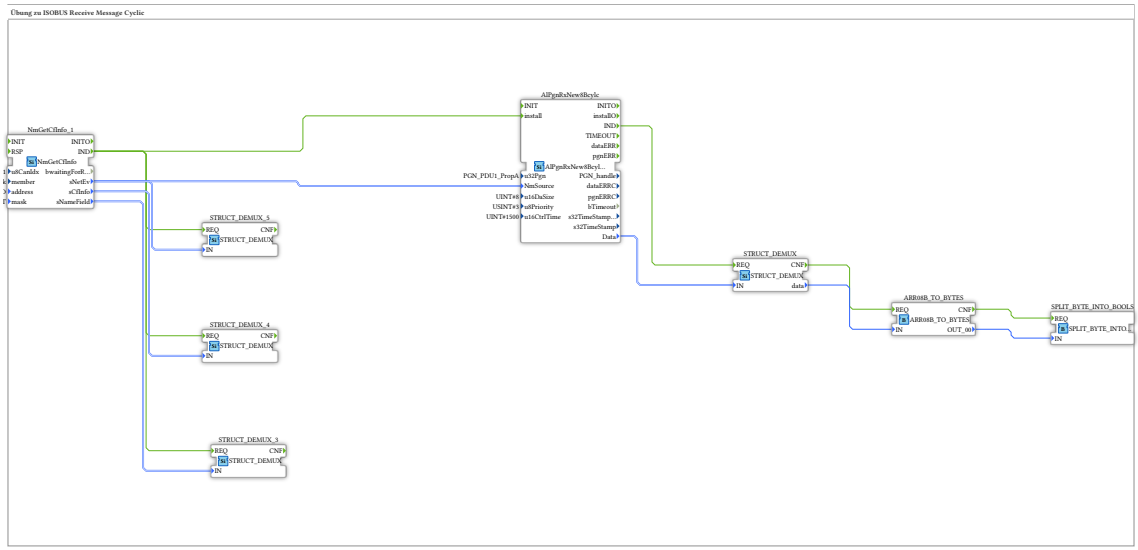
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_130_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.439 Uebung_131_AX: Übung zu ISOBUS Receive Message Cyclic

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_131_AX (Übung zu ISOBUS Receive Message Cyclic).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Übung zu ISOBUS Receive Message Cyclic**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_131_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **STRUCT_DEMUX_3**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **NmGetCfInfo_1**: Instanz des Typs isobus::pgn::NmGetCfInfo.
  - Parameter u8CanIdx = NODE1
  - Parameter member = network
  - Parameter address = PEAK_ADD
  - Parameter mask = PEAK_FLT
- **STRUCT_DEMUX_4**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **STRUCT_DEMUX_5**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **AlPgnRxNew8Bcylc**: Instanz des Typs isobus::pgn::rx::AlPgnRxNew8Bcylc.

  - Parameter u32Pgn = PGN_PDU1_PropA
  - Parameter u16DaSize = 8
  - Parameter u8Priority = 3
  - Parameter u16CtrlTime = 1500
- **STRUCT_DEMUX**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **ARR08B_TO_BYTES**: Instanz des Typs logiBUS::utils::conversion::arr::forwarding::ARR08B_TO_BYTES.
- **SPLIT_BYTE_INTO_BOOLS**: Instanz des Typs eclipse4diac::utils::splitting::SPLIT_BYTE_INTO_BOOLS.

Verbindungen und Schnittstellen

Ereignisverbindungen:

- NmGetCfInfo_1.IND -> STRUCT_DEMUX_3.REQ
- NmGetCfInfo_1.IND -> STRUCT_DEMUX_4.REQ
- NmGetCfInfo_1.IND -> STRUCT_DEMUX_5.REQ
- NmGetCfInfo_1.IND -> AlPgnRxNew8Bcylc.install
- AlPgnRxNew8Bcylc.IND -> STRUCT_DEMUX.REQ
- STRUCT_DEMUX.CNF -> ARR08B_TO_BYTES.REQ
- ARR08B_TO_BYTES.CNF -> SPLIT_BYTE_INTO_BOOLS.REQ

Datenverbindungen:

- NmGetCfInfo_1.sNameField -> STRUCT_DEMUX_3.IN
- NmGetCfInfo_1.sNetEv -> STRUCT_DEMUX_5.IN
- NmGetCfInfo_1.sCfInfo -> STRUCT_DEMUX_4.IN
- NmGetCfInfo_1.sNetEv -> AlPgnRxNew8Bcylc.NmSource
- AlPgnRxNew8Bcylc.Data -> STRUCT_DEMUX.IN
- STRUCT_DEMUX.data -> ARR08B_TO_BYTES.IN
- ARR08B_TO_BYTES.OUT_00 -> SPLIT_BYTE_INTO_BOOLS.IN

Programmablauf und Funktionsweise

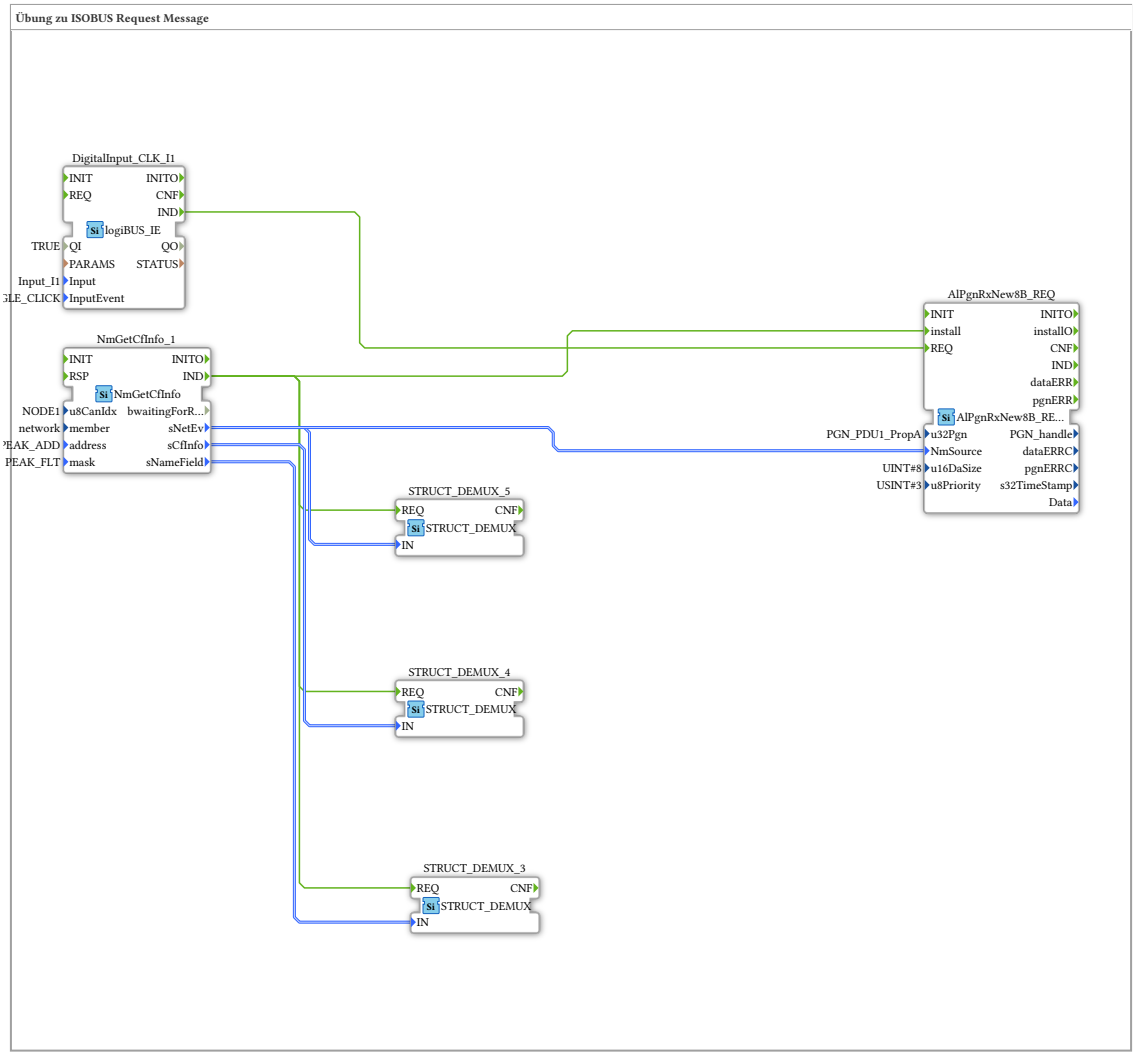
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_131_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.440 Uebung_132_AX: Übung zu ISOBUS Request Message

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_132_AX (Übung zu ISOBUS Request Message).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Übung zu ISOBUS Request Message**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_132_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **STRUCT_DEMUX_3**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **NmGetCfInfo_1**: Instanz des Typs isobus::pgn::NmGetCfInfo.
  - Parameter u8CanIdx = NODE1
  - Parameter member = network
  - Parameter address = PEAK_ADD
  - Parameter mask = PEAK_FLT
- **STRUCT_DEMUX_4**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **STRUCT_DEMUX_5**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **AlPgnRxNew8B_REQ**: Instanz des Typs isobus::pgn::rx::AlPgnRxNew8B_REQ.
  - Parameter u32Pgn = PGN_PDU1_PropA
  - Parameter u16DaSize = 8
  - Parameter u8Priority = 3
- **DigitalInput_CLK_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1

– Parameter InputEvent = BUTTON_SINGLE_CLICK

Verbindungen und Schnittstellen

Ereignisverbindungen:

- NmGetCfInfo_1.IND -> STRUCT_DEMUX_3.REQ
- NmGetCfInfo_1.IND -> STRUCT_DEMUX_4.REQ
- NmGetCfInfo_1.IND -> STRUCT_DEMUX_5.REQ
- NmGetCfInfo_1.IND -> AlPgnRxNew8B_REQ.install
- DigitalInput_CLK_I1.IND -> AlPgnRxNew8B_REQ.REQ

Datenverbindungen:

- NmGetCfInfo_1.sNameField -> STRUCT_DEMUX_3.IN
- NmGetCfInfo_1.sNetEv -> STRUCT_DEMUX_5.IN
- NmGetCfInfo_1.sCfInfo -> STRUCT_DEMUX_4.IN
- NmGetCfInfo_1.sNetEv -> AlPgnRxNew8B_REQ.NmSource

Programmablauf und Funktionsweise

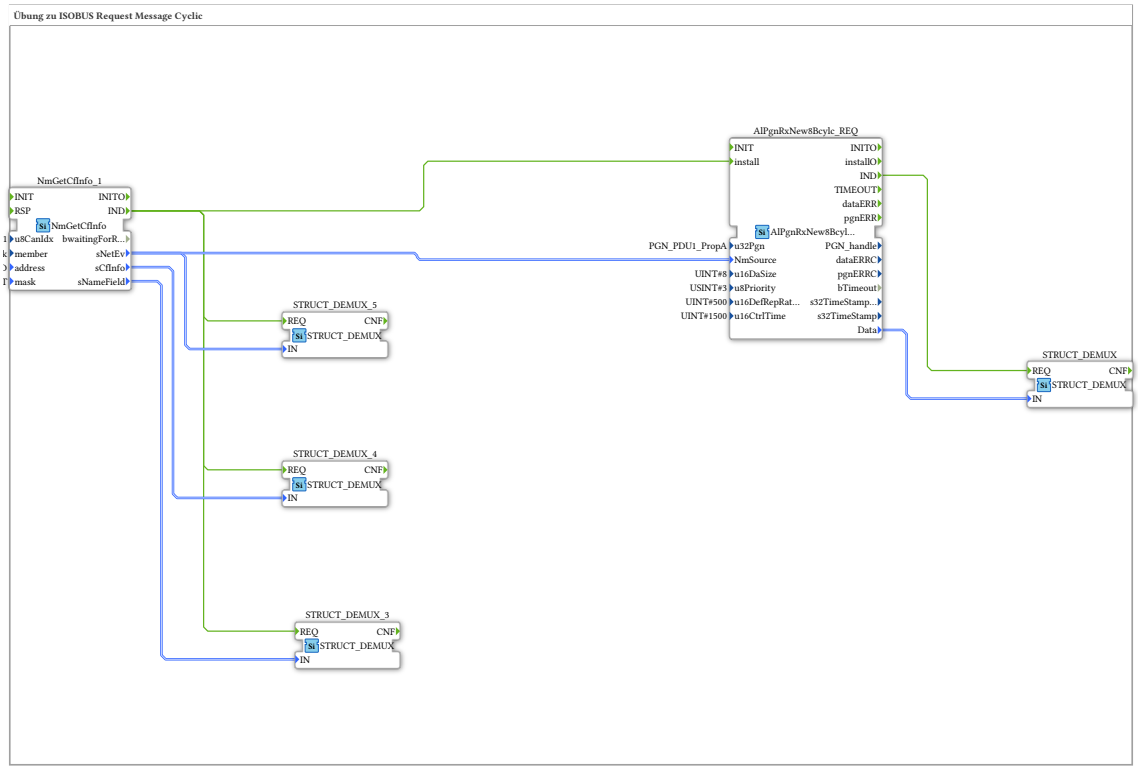
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_132_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.441 Uebung_133_AX: Übung zu ISOBUS Request Message Cyclic

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_133_AX (Übung zu ISOBUS Request Message Cyclic).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Übung zu ISOBUS Request Message Cyclic**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_133_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **STRUCT_DEMUX_3**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **NmGetCfInfo_1**: Instanz des Typs isobus::pgn::NmGetCfInfo.
  - Parameter u8CanIdx = NODE1
  - Parameter member = network
  - Parameter address = PEAK_ADD
  - Parameter mask = PEAK_FLT
- **STRUCT_DEMUX_4**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **STRUCT_DEMUX_5**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **AlPgnRxNew8Bcylc_REQ**: Instanz des Typs isobus::pgn::rx::AlPgnRxNew8Bcylc_REQ.
  - Parameter u32Pgn = PGN_PDU1_PropA
  - Parameter u16DaSize = 8
  - Parameter u8Priority = 3
  - Parameter u16DefRepRate = 500
  - Parameter u16CtrlTime = 1500
- **STRUCT_DEMUX**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.

Verbindungen und Schnittstellen

Ereignisverbindungen:

- NmGetCfInfo_1.IND -> STRUCT_DEMUX_3.REQ
- NmGetCfInfo_1.IND -> STRUCT_DEMUX_4.REQ
- NmGetCfInfo_1.IND -> STRUCT_DEMUX_5.REQ
- NmGetCfInfo_1.IND -> AlPgnRxNew8Bcylc_REQ.install
- AlPgnRxNew8Bcylc_REQ.IND -> STRUCT_DEMUX.REQ

Datenverbindungen:

- NmGetCfInfo_1.sNameField -> STRUCT_DEMUX_3.IN
- NmGetCfInfo_1.sNetEv -> STRUCT_DEMUX_5.IN
- NmGetCfInfo_1.sCfInfo -> STRUCT_DEMUX_4.IN
- NmGetCfInfo_1.sNetEv -> AlPgnRxNew8Bcylc_REQ.NmSource
- AlPgnRxNew8Bcylc_REQ.Data -> STRUCT_DEMUX.IN

Programmablauf und Funktionsweise

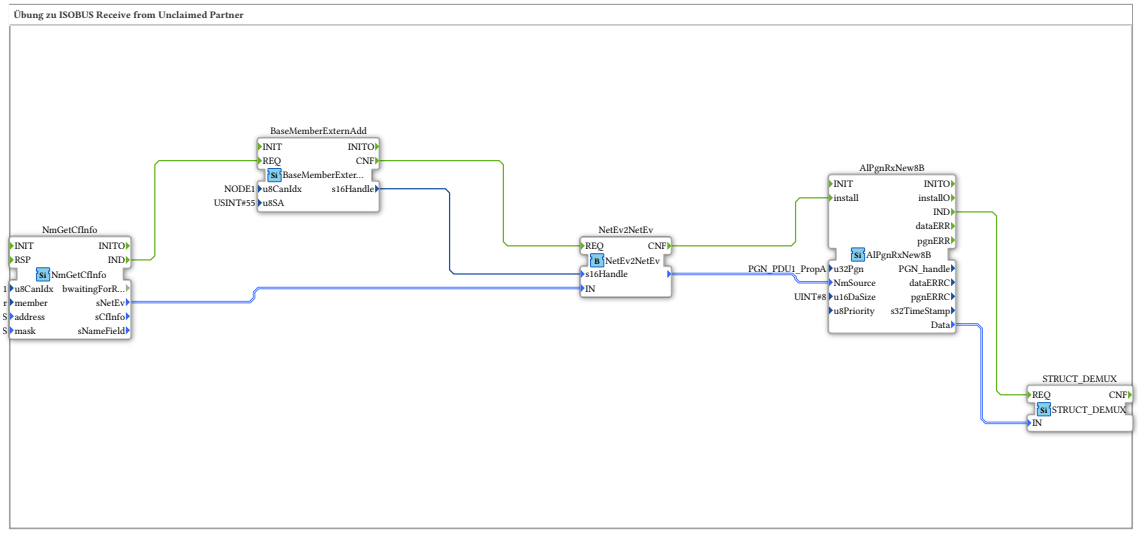
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_133_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.442 Uebung_134_AX: Übung zu ISOBUS Receive from Unclaimed Partner

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_134_AX (Übung zu ISOBUS Receive from Unclaimed Partner).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Übung zu ISOBUS Receive from Unclaimed Partner**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_134_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **BaseMemberExternAdd**: Instanz des Typs isobus::pgn::BaseMemberExternAdd.
  - Parameter u8CanIdx = NODE1
  - Parameter u8SA = 55
- **AlPgnRxNew8B**: Instanz des Typs isobus::pgn::rx::AlPgnRxNew8B.
  - Parameter u32Pgn = PGN_PDU1_PropA
  - Parameter u16DaSize = 8
- **NmGetCfInfo**: Instanz des Typs isobus::pgn::NmGetCfInfo.
  - Parameter u8CanIdx = NODE1
  - Parameter member = thisMember
  - Parameter address = ADD_ALL_PASS
  - Parameter mask = FLT_ALL_PASS
- **NetEv2NetEv**: Instanz des Typs isobus::pgn::NetEv2NetEv.
- **STRUCT_DEMUX**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.

Verbindungen und Schnittstellen

Ereignisverbindungen:

- BaseMemberExternAdd.CNF -> NetEv2NetEv.REQ
- NetEv2NetEv.CNF -> AlPgnRxNew8B.install
- NmGetCfInfo.IND -> BaseMemberExternAdd.REQ
- AlPgnRxNew8B.IND -> STRUCT_DEMUX.REQ

Datenverbindungen:

- BaseMemberExternAdd.s16Handle -> NetEv2NetEv.s16Handle
- NmGetCfInfo.sNetEv -> NetEv2NetEv.IN
- NetEv2NetEv. -> AlPgnRxNew8B.NmSource
- AlPgnRxNew8B.Data -> STRUCT_DEMUX.IN

Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.

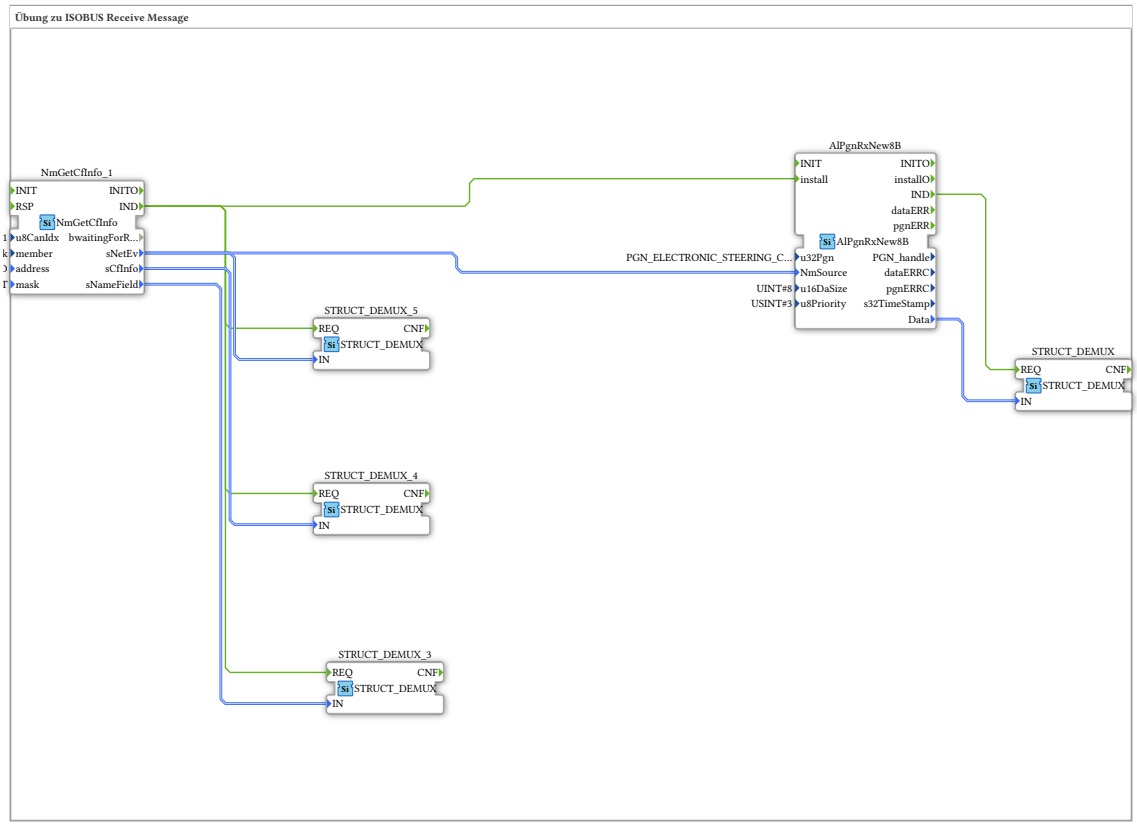
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_134_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.443 Uebung_135_AX: Übung zu ISOBUS Receive Message

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_135_AX (Übung zu ISOBUS Receive Message).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Übung zu ISOBUS Receive Message**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_135_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **STRUCT_DEMUX_3**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **NmGetCfInfo_1**: Instanz des Typs isobus::pgn::NmGetCfInfo.
  - Parameter u8CanIdx = NODE1
  - Parameter member = network
  - Parameter address = PRIM_TECU_ADD
  - Parameter mask = PRIM_TECU_FLT
- **STRUCT_DEMUX_4**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **STRUCT_DEMUX_5**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.
- **AlPgnRxNew8B**: Instanz des Typs isobus::pgn::rx::AlPgnRxNew8B.
  - Parameter u32Pgn = PGN_ELECTRONIC_STEERING_CONTROL

  - Parameter u16DaSize = 8
  - Parameter u8Priority = 3
- **STRUCT_DEMUX**: Instanz des Typs eclipse4diac::convert::STRUCT_DEMUX.

Verbindungen und Schnittstellen

Ereignisverbindungen:

- NmGetCfInfo_1.IND -> STRUCT_DEMUX_3.REQ
- NmGetCfInfo_1.IND -> STRUCT_DEMUX_4.REQ
- NmGetCfInfo_1.IND -> STRUCT_DEMUX_5.REQ
- NmGetCfInfo_1.IND -> AlPgnRxNew8B.install
- AlPgnRxNew8B.IND -> STRUCT_DEMUX.REQ

Datenverbindungen:

- NmGetCfInfo_1.sNameField -> STRUCT_DEMUX_3.IN
- NmGetCfInfo_1.sNetEv -> STRUCT_DEMUX_5.IN
- NmGetCfInfo_1.sCfInfo -> STRUCT_DEMUX_4.IN
- NmGetCfInfo_1.sNetEv -> AlPgnRxNew8B.NmSource
- AlPgnRxNew8B.Data -> STRUCT_DEMUX.IN

Programmablauf und Funktionsweise

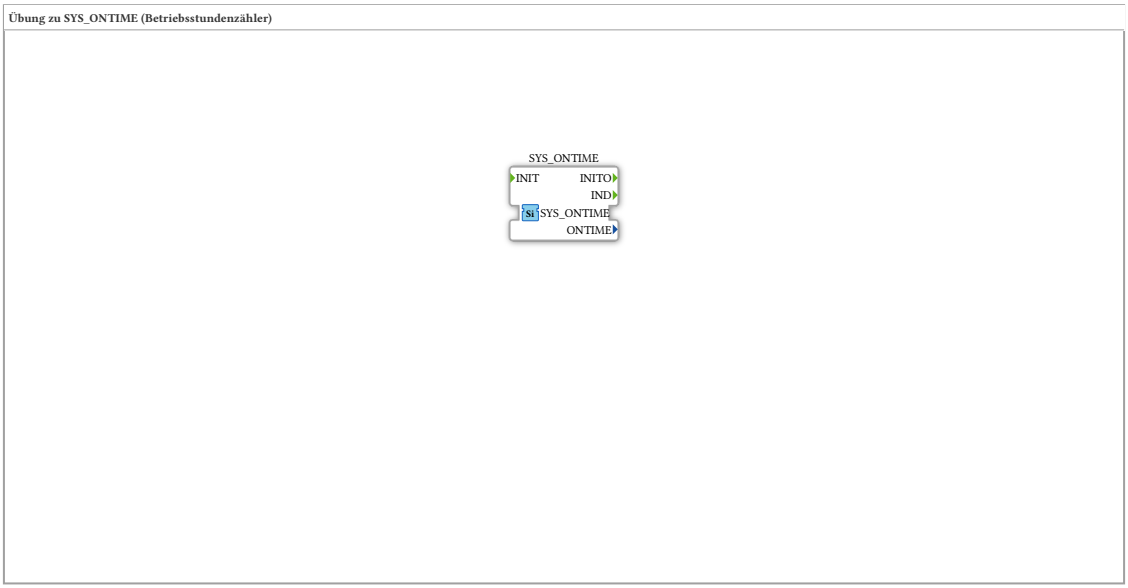
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_135_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.444 Uebung_140_AX: Übung zu SYS_ONTIME (Betriebsstundenzähler)

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_140_AX (Übung zu SYS_ONTIME (Betriebsstundenzähler)).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Übung zu SYS_ONTIME (Betriebsstundenzähler)**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_140_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **SYS_ONTIME**: Instanz des Typs logiBUS::signalprocessing::measurement::SYS_ONTIME.

Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Signalweiterleitung**: Die Bausteine verarbeiten Daten- und Steuersignale gemäß ihrer Konfiguration.

Zusammenfassung

Die Übung Uebung_140_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.445 Uebung_150_AX: Impulszähler & DigitalInput_I1 auf DigitalOutput_Q1, mit Plug and Socket



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_150_AX`. Hier nutzen wir den schnellen Zähleingang der Steuerung.

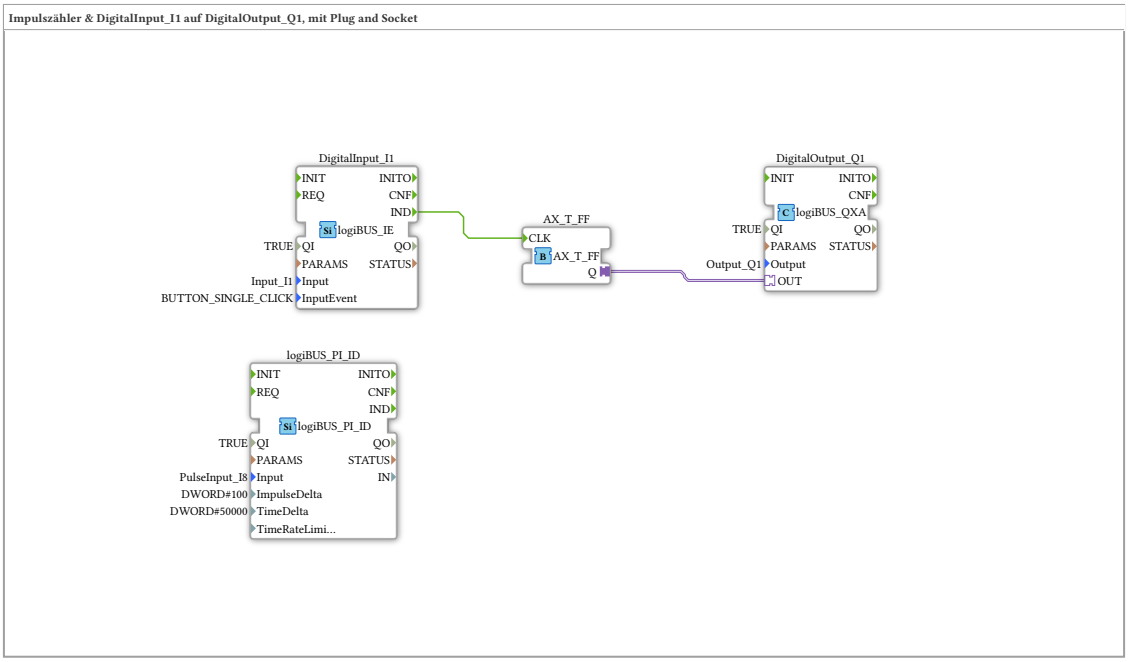
Ziel der Übung

Erfassung von schnellen Impulsen (z.B. Drehzahl, Durchfluss).

Beschreibung und Komponenten

Die Subapplikation `Uebung_150_AX.SUB` kombiniert eine Standard-Beleuchtungslogik mit einem Impulszähler-Baustein.

Funktionsbausteine (FBs)



- **`logiBUS_PI_ID`**: Typ `PulseInput_ID`. Erfasst Impulse am Hardware-Eingang `I8`.
- **`DigitalInput_I1`**: Taster für die Lampe.
- **`AX_T_FF`**: Toggle für die Lampe.

Funktionsweise

Der Baustein `logiBUS_PI_ID` arbeitet im Hintergrund. Er zählt die Impulse am Eingang `I8`.

- `ImpulseDelta = 100`: Der Baustein meldet sich (sendet ein Event), wenn 100 neue Impulse gezählt wurden.
- `TimeDelta = 50000` (µs): Oder wenn 50ms vergangen sind.

Dies ermöglicht die Erfassung von Hochgeschwindigkeitssignalen, die für normale digitale Eingänge zu schnell wären. Die restliche Schaltung (`I1` auf `Q1`) läuft völlig unabhängig davon weiter.

Anwendungsbeispiel

Radarsensor / Geschwindigkeitsmessung: Ein Sensor am Rad liefert Impulse. Die Steuerung zählt diese, um die Fahrgeschwindigkeit des Traktors zu berechnen.

2.1.2.446 Uebung_151_AX: Impulszähler & DigitalInput_I1 auf DigitalOutput_Q1, mit Plug and Socket



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_151_AX`.

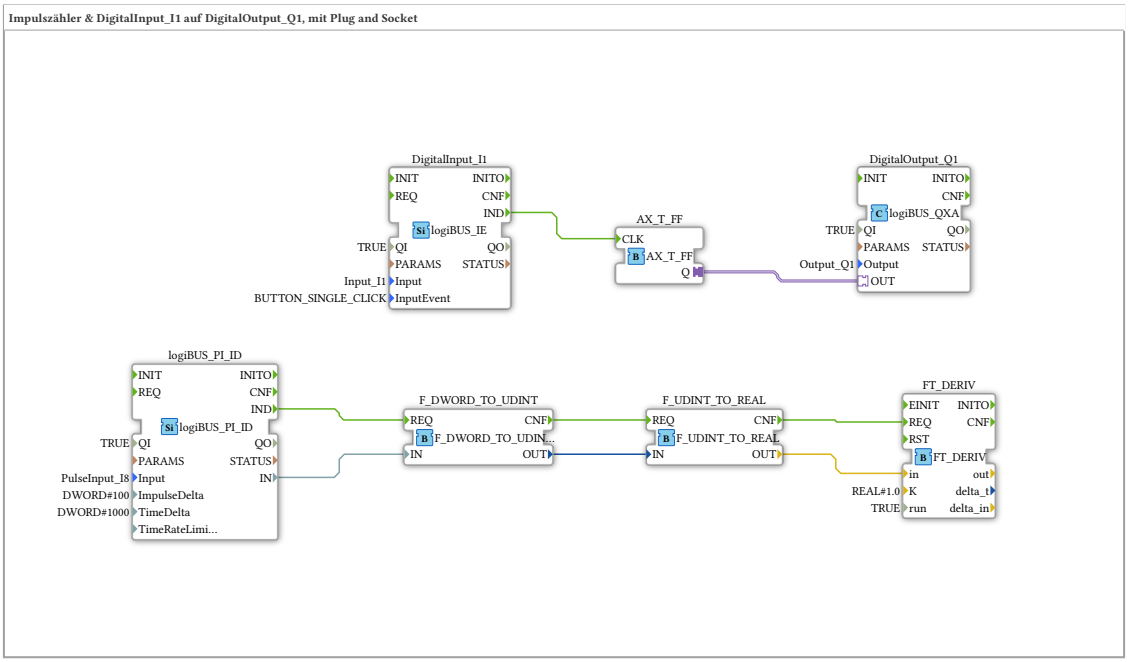
Ziel der Übung

Berechnung einer zeitlichen Änderung (Differenzialquotient) aus Impulswerten.

Beschreibung und Komponenten

Die Subapplikation `Uebung_151_AX.SUB` erweitert den Impulszähler um mathematische Funktionen.

Funktionsbausteine (FBs)



- **`logiBUS_PI_ID`**: Liefert den aktuellen Zählerstand.
- **`FT_DERIV`**: Ein Baustein aus der **OSCAT** Bibliothek zur Berechnung der Ableitung (Änderungsrate).

Funktionsweise

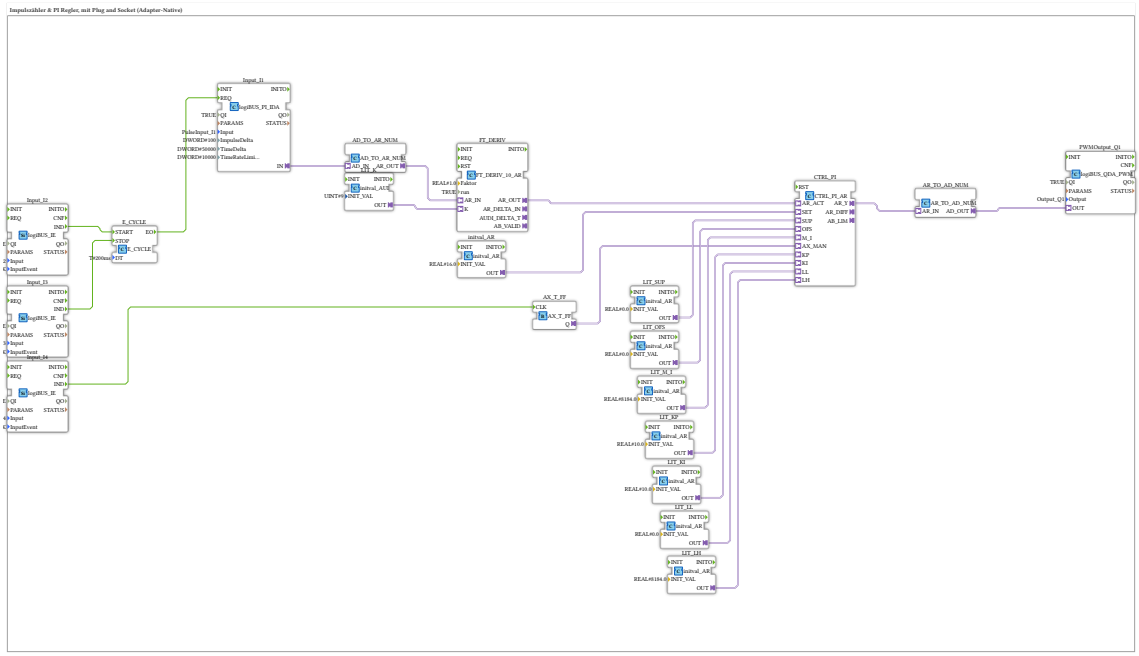
1. Der Zählerstand (DWORD) wird in eine Fließkommazahl (REAL) gewandelt.

2. Der `FT_DERIV` Baustein berechnet, wie schnell sich dieser Wert über die Zeit ändert.
3. Das Ergebnis ist direkt proportional zur Frequenz der Eingangsimpulse (z.B. km/h oder U/min).

Anwendungsbeispiel

Drehzahlüberwachung: Ein Sensor am Lüfterrad liefert Impulse. Steigt oder fällt die Änderung pro Sekunde unter einen Schwellwert, kann ein Alarm ausgelöst werden.

2.1.2.447 Uebung_152_AX: Impulszähler und PI-Regler mit Plug and Socket (adapternativ)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung misst Impulse an `I1`, bildet daraus die Ableitung (`FT_DERIV_10_AR`), regelt diese mit einem PI-Regler (`CTRL_PI_AR`) auf einen Sollwert und gibt den Stellwert als PWM-Signal an `Q1` aus. Die gesamte Signalkette ist adapternativ aufgebaut: Istwert, Sollwert, Handumschaltung und alle Reglerparameter werden über Adapter-Sockets zugeführt, die festen Werte kommen aus `initval_AR`-/`initval_AUI`-Konstantenbausteinen.

Verwendete Funktionsbausteine (FBs)

- **Input_I2**: Klick-Eingang: Messung starten
  - **Typ**: `logiBUS::io::DI::logiBUS_IE`
  - **Parameter**: `QI = TRUE`, `Input = Input_I2`, `InputEvent = BUTTON_SINGLE_CLICK`
  - **Erklärung**: Meldet einen Einzelklick auf `I2` als Event `IND`.
- **Input_I3**: Klick-Eingang: Messung stoppen
  - **Typ**: `logiBUS::io::DI::logiBUS_IE`
  - **Parameter**: `QI = TRUE`, `Input = Input_I3`, `InputEvent = BUTTON_SINGLE_CLICK`
  - **Erklärung**: Meldet einen Einzelklick auf `I3` als Event `IND`.
- **E_CYCLE**: Abtasttakt
  - **Typ**: `iec61499::events::E_CYCLE`
  - **Parameter**: `DT = T#200ms`
  - **Erklärung**: Erzeugt nach `START` alle 200 ms ein Event `EO`, bis `STOP` eintrifft.
- **Input_I1**: Impulseingang
  - **Typ**: `logiBUS::io::PI::logiBUS_PI_IDA`
  - **Parameter**: `QI = TRUE`, `Input = PulseInput_I1`, `ImpulseDelta = 100`, `TimeDelta = 50000`, `TimeRateLimit = 10000`
  - **Erklärung**: Zählt die Impulse an `I1` und gibt sie bei jeder Anfrage (`REQ`) über den Adapter `IN` aus.

- **AD_TO_AR_NUM**: Adapter-Konvertierung
  - **Typ**: `adapter::conversion::unidirectional::AD_TO_AR_NUM`
  - **Parameter**: keine
  - **Erklärung**: Wandelt den `AD`-Adapterwert in einen `AR`-Adapterwert (REAL) um.
- **LIT_K**: Konstante für das Fenster der Ableitung
  - **Typ**: `adapter::types::unidirectional::AUI::initval::initval_AUI`
  - **Parameter**: `INIT_VAL = 9`
  - **Erklärung**: Liefert die Konstante `9` als `AUI`-Adapterwert an `FT_DERIV.K`.
- **FT_DERIV**: Ableitung (Drehzahl aus Impulsen)
  - **Typ**: `OSCAT_adapter::Control::FT_DERIV_10_AR`
  - **Parameter**: `Faktor = 1.0`, `run = TRUE`
  - **Erklärung**: AR-Adapter-Wrapper um OSCAT `FT_DERIV_10`: bildet aus dem Eingangssignal (`AR_IN`) die Ableitung. Das Fenster wird über den Adapter `K` gewählt (`0` = 1 Periode, `9` = 10 Perioden).
- **Input_I4**: Klick-Eingang: Hand/Automatik
  - **Typ**: `logiBUS::io::DI::logiBUS_IE`
  - **Parameter**: `QI = TRUE`, `Input = Input_I4`, `InputEvent = BUTTON_SINGLE_CLICK`
  - **Erklärung**: Meldet einen Einzelklick auf `I4` als Event `IND`.
- **AX_T_FF**: Toggle-Flip-Flop
  - **Typ**: `adapter::events::unidirectional::AX_T_FF`
  - **Parameter**: keine
  - **Erklärung**: Wechselt mit jedem `CLK`-Event den Zustand von `Q`.
- **initval_AR**: Sollwert
  - **Typ**: `adapter::types::unidirectional::AR::initval::initval_AR`
  - **Parameter**: `INIT_VAL = 16.0`
  - **Erklärung**: Liefert den Sollwert `16.0` an `CTRL_PI.SET`.
- **LIT_SUP, LIT_OFS, LIT_M_I, LIT_KP, LIT_KI, LIT_LL, LIT_LH**: Konstanten für die Reglerparameter
  - **Typ**: `adapter::types::unidirectional::AR::initval::initval_AR`
  - **Parameter**: `INIT_VAL = 0.0` (`LIT_SUP`), `0.0` (`LIT_OFS`), `8184.0` (`LIT_M_I`), `10.0` (`LIT_KP`), `10.0` (`LIT_KI`), `0.0` (`LIT_LL`), `8184.0` (`LIT_LH`)
  - **Erklärung**: Liefern als `AR`-Adapterwerte: Rauschunterdrückung `SUP`, Offset `OFS`, Handwert `M_I`, Proportionalbeiwert `KP`, Integralbeiwert `KI`, unteren Grenzwert `LL` und oberen Grenzwert `LH` an `CTRL_PI`.
- **CTRL_PI**: PI-Regler
  - **Typ**: `OSCAT_adapter::Control::CTRL_PI_AR`
  - **Parameter**: keine
  - **Erklärung**: AR-Adapter-Wrapper um OSCAT `CTRL_PI`: Istwert (`AR_ACT`), Sollwert (`SET`), Handumschaltung (`AX_MAN`) und alle Reglerparameter sind Adapter-Sockets; der Stellwert liegt am Adapter `AR_Y`.
- **AR_TO_AD_NUM**: Adapter-Konvertierung
  - **Typ**: `adapter::conversion::unidirectional::AR_TO_AD_NUM`
  - **Parameter**: keine
  - **Erklärung**: Wandelt den `AR`-Stellwert in einen `AD`-Adapterwert um.
- **PWMOutput_Q1**: PWM-Ausgang
  - **Typ**: `logiBUS::io::DQ::logiBUS_QDA_PWM`
  - **Parameter**: `QI = TRUE`, `Output = Output_Q1`
  - **Erklärung**: Gibt den Stellwert als PWM-Signal am physischen Ausgang `Q1` aus.

Programmablauf und Verbindungen

1. `Input_I2.IND` ⟶ `E_CYCLE.START` und `Input_I3.IND` ⟶ `E_CYCLE.STOP` (EventConnections): `I2` startet, `I3` stoppt den 200-ms-Takt.
2. `E_CYCLE.EO` ⟶ `Input_I1.REQ` (EventConnection): Mit jedem Takt wird der Impulseingang abgefragt.
3. `Input_I1.IN` ⟶ `AD_TO_AR_NUM.AD_IN` und `AD_TO_AR_NUM.AR_OUT` ⟶ `FT_DERIV.AR_IN` (AdapterConnections): Der Impulswert wird in einen `AR`-Wert gewandelt und der Ableitung zugeführt. `LIT_K.OUT` ⟶ `FT_DERIV.K` legt das Fenster fest.
4. `FT_DERIV.AR_OUT` ⟶ `CTRL_PI.AR_ACT` (AdapterConnection): Die Ableitung ist der Istwert des Reglers; `initval_AR.OUT` ⟶ `CTRL_PI.SET` liefert den Sollwert `16.0`.
5. `LIT_SUP.OUT`, `LIT_OFS.OUT`, `LIT_M_I.OUT`, `LIT_KP.OUT`, `LIT_KI.OUT`, `LIT_LL.OUT`, `LIT_LH.OUT` ⟶ `CTRL_PI.SUP/OFS/M_I/KP/KI/LL/LH` (AdapterConnections): Die Reglerparameter kommen als Konstanten-Adapter.

6. `Input_I4.IND` ⟶ `AX_T_FF.CLK` (EventConnection) und `AX_T_FF.Q` ⟶ `CTRL_PI.AX_MAN` (AdapterConnection): Jeder Klick auf `I4` schaltet die Handumschaltung des Reglers um.
7. `CTRL_PI.AR_Y` ⟶ `AR_TO_AD_NUM.AR_IN` und `AR_TO_AD_NUM.AD_OUT` ⟶ `PWMOutput_Q1.OUT` (AdapterConnections): Der Stellwert wird gewandelt und als PWM an `Q1` ausgegeben.

Verhalten: Nach einem Klick auf `I2` wird alle 200 ms der Impulseingang abgefragt, bis `I3` den Takt stoppt. Der Regler arbeitet auf den Sollwert `16.0`; `I4` schaltet die Handumschaltung des Reglers (`AX_MAN`) um. Der Stellwert ist auf `LL = 0.0` bis `LH = 8184.0` begrenzt.

Zusammenfassung

Die Übung zeigt einen Regelkreis, der vollständig mit Plug und Socket (Adaptern) verdrahtet ist. Seit Version 1.3 sind die früheren Parameter an `CTRL_PI` und `FT_DERIV` durch Konstanten-Bausteine (`initval_AR`/`initval_AUI`) und Adapterverbindungen ersetzt, weil `CTRL_PI_AR` und `FT_DERIV_10_AR` diese Eingänge nur noch als Adapter-Sockets besitzen.

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

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

2.1.2.448 Übung 152 AX: Impulszähler & PI-Regler mit Adapter-Kopplung

Thema: Drehzahl-/Frequenzregelung mit Impulszähler und PI-Regler

Situationsbeschreibung

In landtechnischen Anwendungen (z. B. Säwellenantrieb, Düngerstreuer-Dosierscheibe, Lüfterdrehzahlsteuerung) muss eine Drehzahl oder Impulsfrequenz gemessen und über einen PI-Regler (`CTRL_PI_AR`) auf einen Sollwert geregelt werden. Die Stellgröße wird als PWM-Signal an einem digitalen Leistungsausgang (`Output_Q1`) ausgegeben.

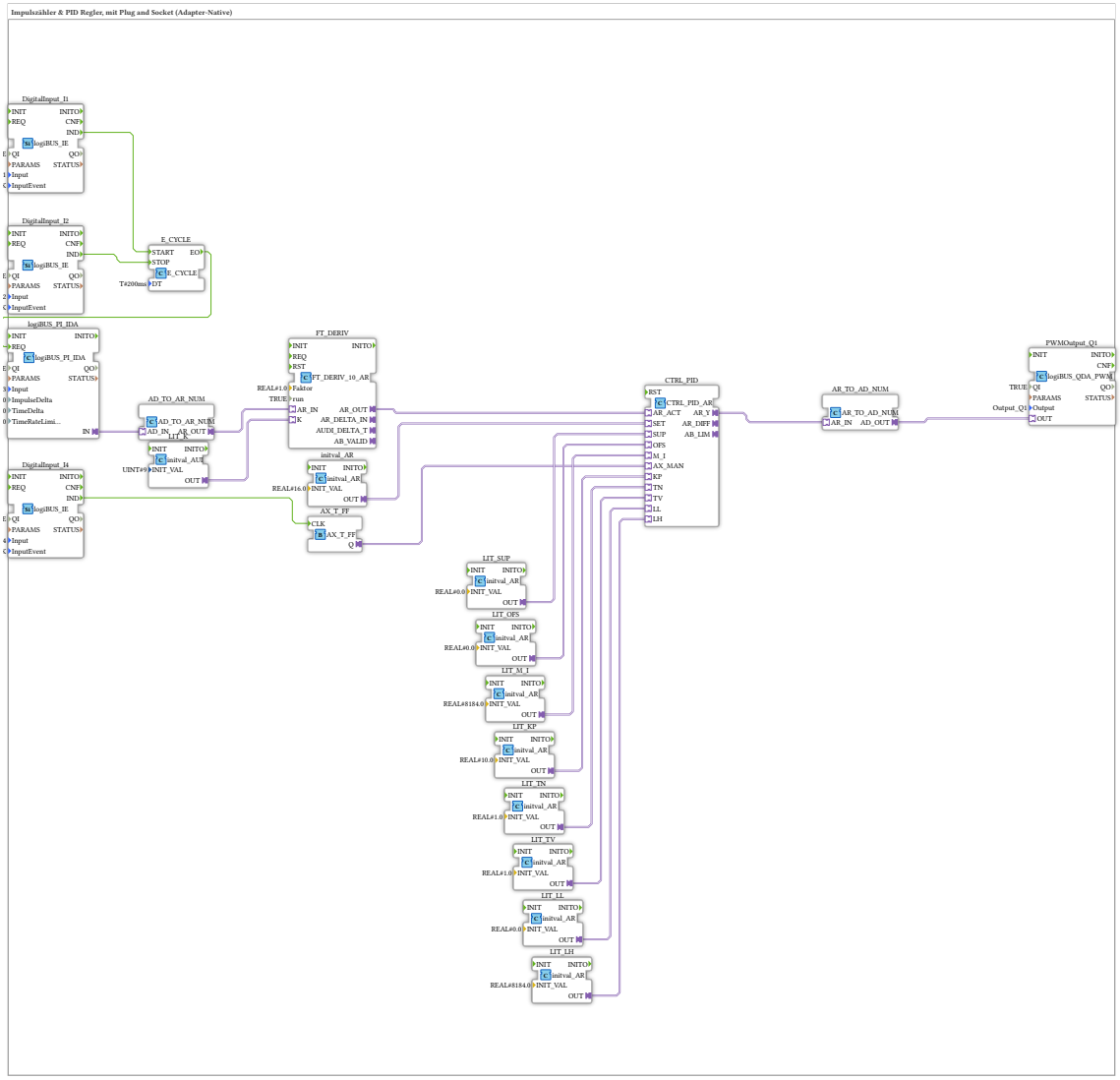
Funktionsbeschreibung

- **Erfassung & Wandlung:** Über den Pulse-Input-Adapterbaustein `logiBUS_PI_IDA` an `PulseInput_I8` werden Zählimpulse zyklisch (getriggert durch `E_CYCLE` alle 200 ms via `I2` Start / `I3` Stopp) erfasst und über `AD_TO_AR_NUM` direkt in das REAL-Adapterformat (`AR`) konvertiert.
- **Signalaufbereitung:** Der `AR`-Adapter streamt an `FT_DERIV_10_AR` (`AR_IN`), wo das Signal differenziert wird, um die aktuelle Frequenz/Drehzahl (`AR_OUT`) zu bestimmen.
- **Regelung:** Der PI-Regler-Adapterwrapper `CTRL_PI_AR` vergleicht den Istwert (`AR_ACT`) mit dem Sollwert (`SET = 16.0`). Über Taster `I1` (via `AX_T_FF.Q`) wird direkt über den Adapter-Socket `AX_MAN` zwischen Automatik- und Handbetrieb umgeschaltet.
- **Stellgrößenausgabe:** Die Stellgröße `AR_Y` des Reglers wird über den Konvertierungsbaustein `AR_TO_AD_NUM` direkt an den PWM-Ausgabe-Baustein `PWMOutput_Q1` (`logiBUS_QDA_PWM`) auf `Output_Q1` übergeben.

Referenzlösung

`Uebung_152_AX.SUB` — validiert in 4diac IDE.

2.1.2.449 Uebung_153_AX: Impulszähler & PID Regler, mit Plug and Socket (Adapternativ)



• — ○ • — ○ • — ○ •

Einleitung

Diese Übung kombiniert einen Impulszähler (`logiBUS_PI_IDA`, Eingang `I3`) mit einem PID-Regler und gibt die Stellgröße als PWM an `Q1` aus. Der Datenfluss läuft vollständig über Adapter (Plug and Socket): `FT_DERIV_10_AR` und `CTRL_PID_AR` stammen aus `OSCAT_adapter`, `AD_TO_AR_NUM` und `AR_TO_AD_NUM` wandeln zwischen den Adaptertypen. Die Reglerparameter (`SET`, `SUP`, `OFS`, `M_I`, `KP`, `TN`, `TV`, `LL`, `LH`) und der Faktor `K` des Differenzierers werden nicht mehr als Parameter, sondern über Literal-Bausteine (`initval_AR`/`initval_AUI`) per Adapter zugeführt. Mit `I1` und `I2` wird die zyklische Abfrage gestartet und gestoppt, mit `I4` wird der Handbetrieb des Reglers umgeschaltet.

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_I1**, **DigitalInput_I2**, **DigitalInput_I4**: Taster mit Klick-Erkennung
  - **Typ**: `logiBUS::io::DI::logiBUS_IE`
  - **Parameter**: `QI = TRUE`, `Input = Input_I1 / Input_I2 / Input_I4`, jeweils `InputEvent = BUTTON_SINGLE_CLICK`
  - **Erklärung**: Melden einen Einzelklick als Event `IND`.
- **E_CYCLE**: Abtasttakt
  - **Typ**: `iec61499::events::E_CYCLE`
  - **Parameter**: `DT = T#200ms`

  - **Erklärung**: Wird durch I1 gestartet (START), durch I2 gestoppt (STOP) und fordert alle 200 ms einen neuen Impulswert an.
- **logiBUS_PI_IDA**: Impulseingang
  - **Typ**: logiBUS::io::PI::logiBUS_PI_IDA
  - **Parameter**: QI = TRUE, Input = PulseInput_I3, ImpulseDelta = 100, TimeDelta = 50000, TimeRateLimit = 10000
  - **Erklärung**: Liefert den Impulswert über den Adapter IN bei jedem REQ.
- **AD_TO_AR_NUM**: Konvertierung AD nach AR
  - **Typ**: adapter::conversion::unidirectional::AD_TO_AR_NUM
  - **Parameter**: keine
  - **Erklärung**: Wandelt den Impulswert (AD_IN) numerisch in einen REAL-Adapterwert (AR_OUT).
- **LIT_K**: Faktor K des Differenzierers
  - **Typ**: adapter::types::unidirectional::AUI::initval::initval_AUI
  - **Parameter**: INIT_VAL = 9
  - **Erklärung**: Liefert K = 9 an FT_DERIV.K.
- **FT_DERIV**: Differenzierer mit Verzögerung
  - **Typ**: OSCAT_adapter::Control::FT_DERIV_10_AR
  - **Parameter**: Faktor = 1.0, run = TRUE
  - **Erklärung**: Bildet aus dem REAL-Wert (AR_IN) die Änderungsrate (AR_OUT), die als Istwert an den Regler geht.
- **AX_T_FF**: Toggle-Flipflop
  - **Typ**: adapter::events::unidirectional::AX_T_FF
  - **Parameter**: keine
  - **Erklärung**: Jeder Klick auf I4 (CLK) kehrt Q um; Q geht an den Eingang AX_MAN des Reglers.
- **initval_AR**: Sollwert
  - **Typ**: adapter::types::unidirectional::AR::initval::initval_AR
  - **Parameter**: INIT_VAL = 16.0
  - **Erklärung**: Liefert den Sollwert an CTRL_PID.SET.
- **CTRL_PID**: PID-Regler
  - **Typ**: OSCAT_adapter::Control::CTRL_PID_AR
  - **Parameter**: keine (alle Eingänge sind Adapter)
  - **Erklärung**: Erhält Istwert (AR_ACT), Sollwert (SET), Handbetrieb (AX_MAN) und die Reglerparameter über Adapter; liefert die Stellgröße AR_Y.
- **LIT_SUP**, **LIT_OFS**, **LIT_M_I**, **LIT_KP**, **LIT_TN**, **LIT_TV**, **LIT_LL**, **LIT_LH**: Literal-Bausteine für die Reglerparameter
  - **Typ**: adapter::types::unidirectional::AR::initval::initval_AR
  - **Parameter**: INIT_VAL = 0.0 (LIT_SUP), 0.0 (LIT_OFS), 8184.0 (LIT_M_I), 10.0 (LIT_KP), 1.0 (LIT_TN), 1.0 (LIT_TV), 0.0 (LIT_LL), 8184.0 (LIT_LH)
  - **Erklärung**: Jeder Baustein speist den gleichnamigen Eingang (SUP, OFS, M_I, KP, TN, TV, LL, LH) von CTRL_PID.
- **AR_TO_AD_NUM**: Konvertierung AR nach AD
  - **Typ**: adapter::conversion::unidirectional::AR_TO_AD_NUM
  - **Parameter**: keine
  - **Erklärung**: Wandelt die Stellgröße (AR_IN) in einen AD-Adapterwert (AD_OUT).
- **PWMOutput_Q1**: PWM-Ausgang
  - **Typ**: logiBUS::io::DQ::logiBUS_QDA_PWM
  - **Parameter**: QI = TRUE, Output = Output_Q1
  - **Erklärung**: Gibt den Wert von AR_TO_AD_NUM.AD_OUT als PWM an Q1 aus.

Programmablauf und Verbindungen

1. DigitalInput_I1.IND → E_CYCLE.START und DigitalInput_I2.IND → E_CYCLE.STOP (EventConnections): I1 startet, I2 stoppt den 200-ms-Takt.
2. E_CYCLE.EO → logiBUS_PI_IDA.REQ (EventConnection): Bei jedem Takt wird der Impulswert angefordert.
3. logiBUS_PI_IDA.IN → AD_TO_AR_NUM.AD_IN, AD_TO_AR_NUM.AR_OUT → FT_DERIV.AR_IN und LIT_K.OUT → FT_DERIV.K (AdapterConnections): Der Impulswert wird gewandelt und differenziert.
4. FT_DERIV.AR_OUT → CTRL_PID.AR_ACT (AdapterConnection): Das Ergebnis ist der Istwert des Reglers.
5. initval_AR.OUT → CTRL_PID.SET (AdapterConnection): Sollwert 16.0.

6. `DigitalInput_I4.IND` ⟶ `AX_T_FF.CLK` und `AX_T_FF.Q` ⟶ `CTRL_PID.AX_MAN` (Event- und AdapterConnection): Jeder Klick auf `I4` schaltet den Handbetrieb des Reglers um.
7. `LIT_SUP.OUT` ⟶ `CTRL_PID.SUP`, `LIT_OFS.OUT` ⟶ `CTRL_PID.OFS`, `LIT_M_I.OUT` ⟶ `CTRL_PID.M_I`, `LIT_KP.OUT` ⟶ `CTRL_PID.KP`, `LIT_TN.OUT` ⟶ `CTRL_PID.TN`, `LIT_TV.OUT` ⟶ `CTRL_PID.TV`, `LIT_LL.OUT` ⟶ `CTRL_PID.LL`, `LIT_LH.OUT` ⟶ `CTRL_PID.LH` (AdapterConnections): Die Reglerparameter.
8. `CTRL_PID.AR_Y` ⟶ `AR_TO_AD_NUM.AR_IN` und `AR_TO_AD_NUM.AD_OUT` ⟶ `PWMOutput_Q1.OUT` (AdapterConnections): Die Stellgröße wird als PWM an `Q1` ausgegeben.

Zusammenfassung

Die Übung 153_AX zeigt einen Regelkreis, bei dem Impulserfassung, Differenzierung und PID-Regler vollständig über Adapter verbunden sind. Reglerparameter und Differenzierfaktor werden dabei nicht mehr als Parameter gesetzt, sondern über `initval_AR`/`initval_AUI`-Literale mit gleichen Werten wie zuvor zugeführt.

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

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

2.1.2.450 Übung 153 AX: Impulszähler & PID-Regler mit Adapter-Kopplung

Thema: Drehzahl-/Frequenzregelung mit Impulszähler und PID-Regler

Situationsbeschreibung

In landtechnischen Anwendungen (z. B. Säwellenantrieb, Düngerstreuer-Dosierscheibe, Lüfterdrehzahlsteuerung) muss eine Drehzahl oder Impulsfrequenz gemessen und über einen PID-Regler (`CTRL_PID_AR`) auf einen Sollwert geregelt werden. Die Stellgröße wird als PWM-Signal an einem digitalen Leistungsausgang (`Output_Q1`) ausgegeben.

Funktionsbeschreibung

- **Erfassung & Wandlung:** Über den Pulse-Input-Adapterbaustein `logiBUS_PI_IDA` an `PulseInput_I8` werden Zählimpulse zyklisch (getriggert durch `E_CYCLE` alle 200 ms via `I2` Start / `I3` Stopp) erfasst und über `AD_TO_AR_NUM` direkt in das REAL-Adapterformat (`AR`) konvertiert.
- **Signalaufbereitung:** Der `AR`-Adapter streamt an `FT_DERIV_10_AR` (`AR_IN`), wo das Signal differenziert wird, um die aktuelle Frequenz/Drehzahl (`AR_OUT`) zu bestimmen.
- **Regelung:** Der PID-Regler-Adapterwrapper `CTRL_PID_AR` vergleicht den Istwert (`AR_ACT`) mit dem Sollwert (`SET = 16.0`). Über Taster `I1` (via `AX_T_FF.Q`) wird direkt über den Adapter-Socket `AX_MAN` zwischen Automatik- und Handbetrieb umgeschaltet.
- **Stellgrößenausgabe:** Die Stellgröße `AR_Y` des Reglers wird über den Konvertierungsbaustein `AR_TO_AD_NUM` direkt an den PWM-Ausgabe-Baustein `PWMOutput_Q1` (`logiBUS_QDA_PWM`) auf `Output_Q1` übergeben.

Referenzlösung

`Uebung_153_AX.SUB` — validiert in 4diac IDE.

2.1.2.451 Uebung_160_AX: Motor Links/Rechtslauf



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_160_AX`.

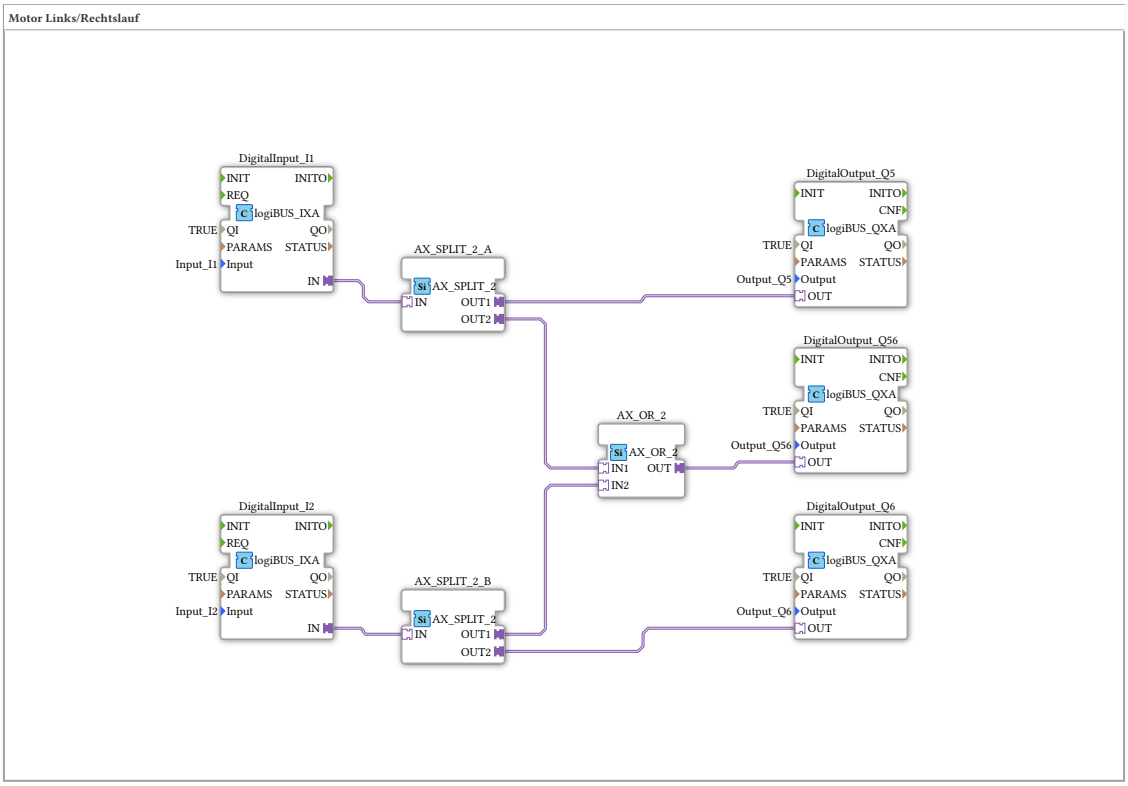
Ziel der Übung

Kombination von Einzelausgängen und einer Sammelmeldung.

Beschreibung und Komponenten

Die Subapplikation `Uebung_160_AX.SUB` steuert zwei Drehrichtungen und einen gemeinsamen Status-Ausgang.

Funktionsbausteine (FBs)



- **`I1`**: Taster für Links (`Q5`).
- **`I2`**: Taster für Rechts (`Q6`).
- **`AX_OR_2`**: Verknüpft beide Signale.
- **`Q56`**: Ein dritter Ausgang.

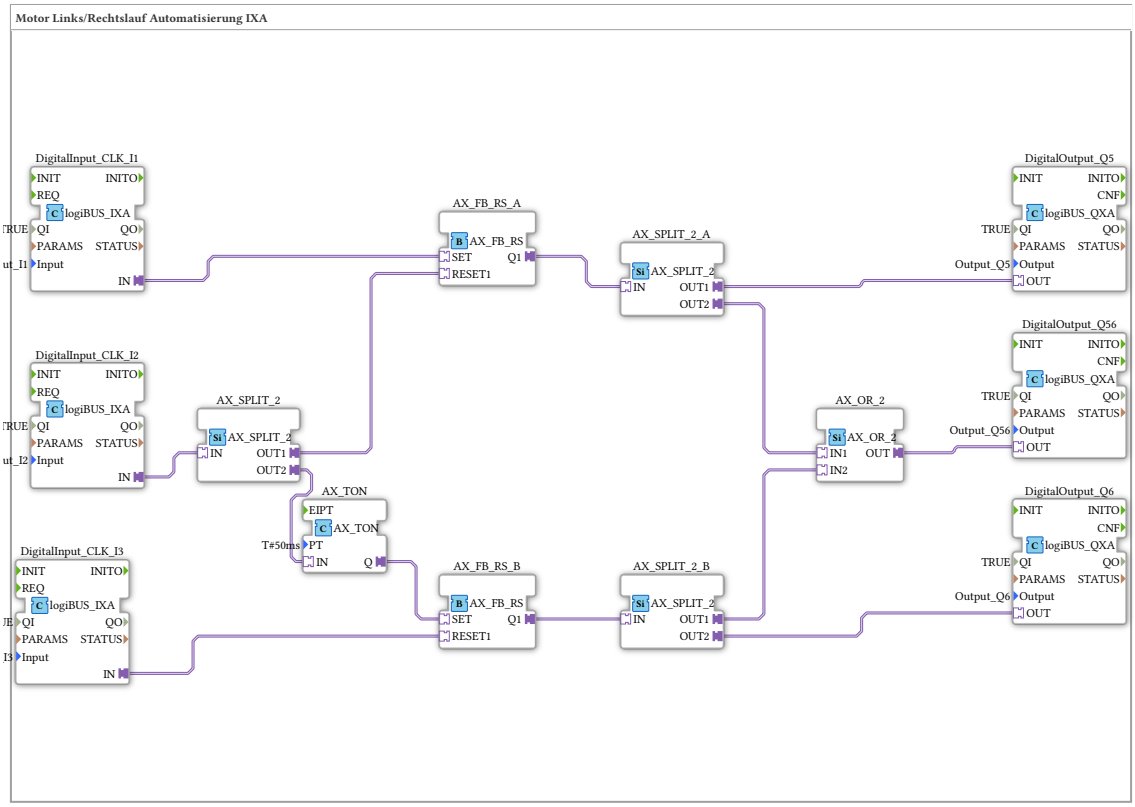
Funktionsweise

1. Drückt man `I1`, geht `Q5` an.
2. Drückt man `I2`, geht `Q6` an.
3. Der Baustein `AX_OR_2` sorgt dafür, dass `Q56` immer dann an ist, wenn **entweder** `Q5` **oder** `Q6` (oder beide) aktiv sind.

Anwendungsbeispiel

Hauptschütz-Ansteuerung: In einer Motorsteuerung sind `Q5` und `Q6` die Richtungsschütze. `Q56` steuert das Hauptschütz an, das in beiden Fällen angezogen sein muss, um den Leistungsteil mit Strom zu versorgen.

2.1.2.452 Uebung_160b2_AX: Motor Links/Rechtslauf Automatisierung IXA



• – ◦ • – ◦ • – ◦ •

Einleitung

Die Übung **Uebung_160b2_AX** realisiert eine Steuerung für einen Motor mit Links- und Rechtslauf-Funktionalität unter Verwendung der Adapter-Technologie (AX/IXA/QXA). Der Fokus liegt auf einer sicheren Umschaltung der Drehrichtung, wobei eine Totzeit (Verzögerung) implementiert ist, um den Motor und die Mechanik beim Wechsel der Drehrichtung zu schützen. Zusätzlich wird der Betriebsstatus über Ausgänge signalisiert.

Verwendete Funktionsbausteine (FBs)

In dieser Sub-Application werden verschiedene Bausteine aus der `logiBUS` und `adapter` Bibliothek verwendet, um die Ein-/Ausgabesignale zu verarbeiten und die logische Verknüpfung herzustellen.

Sub-Bausteine: E/A-Schnittstellen (logiBUS)

Diese Bausteine stellen die Verbindung zur physischen Hardware her.

- **Typ**: `logiBUS::io::DI::logiBUS_IXA` (Eingänge) und `logiBUS::io::DQ::logiBUS_QXA` (Ausgänge)
- **Verwendete interne FBs**:
  - **DigitalInput_CLK_I1**: `logiBUS_IXA`
    - Parameter: `Input` = `Input_I1`
    - Funktion: Startsignal für Drehrichtung 1 (z.B. Links).
  - **DigitalInput_CLK_I2**: `logiBUS_IXA`
    - Parameter: `Input` = `Input_I2`
    - Funktion: Umschalt-Signal / Stoppt Richtung 1 und startet Richtung 2 verzögert.
  - **DigitalInput_CLK_I3**: `logiBUS_IXA`
    - Parameter: `Input` = `Input_I3`
    - Funktion: Stoppsignal für Drehrichtung 2.
  - **DigitalOutput_Q5**: `logiBUS_QXA`
    - Parameter: `Output` = `Output_Q5`
    - Funktion: Ansteuerung Schütz/Motor Drehrichtung 1.
  - **DigitalOutput_Q6**: `logiBUS_QXA`

    - Parameter: `Output` = `Output_Q6`
    - Funktion: Ansteuerung Schütz/Motor Drehrichtung 2.
  - **DigitalOutput_Q56**: `logiBUS_QXA`
    - Parameter: `Output` = `Output_Q56`
    - Funktion: Sammelanzeige „Motor läuft“ (Richtung 1 oder 2).

Sub-Bausteine: Logik und Speicher (Adapter)

Diese Bausteine verarbeiten die Signale logisch.

- **Typ**: `adapter::iec61131::bistableElements::AX_FB_RS`
- **Verwendete interne FBs**:
  - **AX_SR_A**: `AX_FB_RS`
    - Funktion: Speicherglied (RS-Flipflop) für Drehrichtung 1.
  - **AX_SR_B**: `AX_FB_RS`
    - Funktion: Speicherglied (RS-Flipflop) für Drehrichtung 2.

Sub-Bausteine: Zeitglieder

- **Typ**: `adapter::events::unidirectional::timers::AX_TON`
- **Verwendete interne FBs**:
  - **AX_TON**: `AX_TON`
    - Parameter: `PT` = `T#50ms`
    - Funktion: Einschaltverzögerung von 50 Millisekunden für den sanften Richtungswechsel.

Sub-Bausteine: Signalverteilung und Verknüpfung

- **Typ**: `adapter::events::unidirectional::AX_SPLIT_2` und `adapter::booleanOperators::AX_OR_2`
- **Verwendete interne FBs**:
  - **AX_SPLIT_2, AX_SPLIT_2_A, AX_SPLIT_2_B**: `AX_SPLIT_2`
    - Funktion: Verteilt ein eingehendes Adaptersignal auf zwei Ausgänge, um parallele Prozesse anzustoßen oder Signale abzweigen zu können.
  - **AX_OR_2**: `AX_OR_2`
    - Funktion: Logisches ODER. Verknüpft die Statusmeldungen beider Drehrichtungen für die Sammelanzeige.

Programmablauf und Verbindungen

Die Schaltung realisiert eine klassische Wendeschützsteuerung mit einer Besonderheit in der Umschaltung durch Tasterverriegelung und Zeitverzögerung.

1. **Start Drehrichtung 1 (Q5):**
   - Das Signal von **Input_I1** setzt den Speicher **AX_SR_A**.
   - Der Ausgang von **AX_SR_A** wird über einen Splitter (**AX_SPLIT_2_A**) direkt auf **Output_Q5** geleitet. Der Motor läuft in Richtung 1.
2. **Umschaltung / Start Drehrichtung 2 (Q6):**
   - Das Signal von **Input_I2** gelangt auf einen Splitter (**AX_SPLIT_2**).
   - **Zweig 1:** Das Signal resettet sofort den Speicher **AX_SR_A**. Damit wird **Output_Q5** unverzüglich abgeschaltet.
   - **Zweig 2:** Das Signal startet den Timer **AX_TON**. Nach Ablauf von 50ms (Parameter `PT`) wird der Speicher **AX_SR_B** gesetzt.
   - Der Ausgang von **AX_SR_B** aktiviert über **AX_SPLIT_2_B** den **Output_Q6**. Der Motor läuft nun in Richtung 2.
   - *Hinweis:* Die 50ms Verzögerung dient als Verriegelungszeit, um einen Kurzschluss zwischen den Phasen bei direkter Umschaltung zu verhindern.
3. **Stopp Drehrichtung 2:**
   - Das Signal von **Input_I3** resettet den Speicher **AX_SR_B**, wodurch **Output_Q6** abschaltet.
4. **Betriebsanzeige (Q56):**
   - Die Signale der beiden Drehrichtungen (von den Splittern A und B kommend) werden im Baustein **AX_OR_2** zusammengeführt.

- Sobald entweder Q5 oder Q6 aktiv ist, wird **Output_Q56** angesteuert. Dies dient als Indikator, dass der Motor in Betrieb ist.

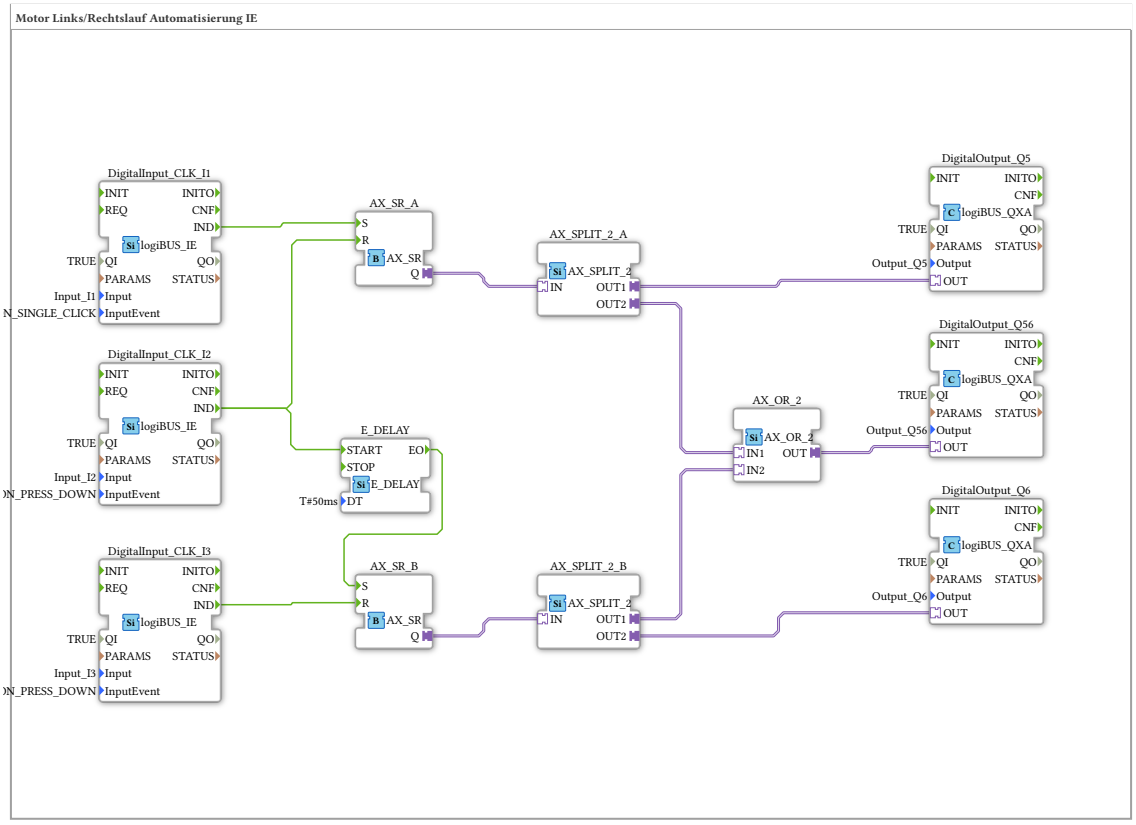
Zusammenfassung

Die `Uebung_160b2_AX` demonstriert eine fortgeschrittene Motorsteuerung unter Verwendung von Adapter-Bausteinen. Sie zeigt, wie man durch den Einsatz von Splittern (`AX_SPLIT_2`) Signale vervielfältigt, um gleichzeitige Aktionen (Reset der einen Seite, Starten des Timers für die andere Seite) auszuführen. Die integrierte Sicherheitslogik mittels `AX_TON` verhindert eine sofortige Richtungsumkehr und schützt somit die angeschlossene Hardware. Die Übung ist ideal, um das Verständnis für sequenzielle Steuerungen und Signalrouting in 4diac zu vertiefen.

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

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

2.1.2.453 Uebung_160b_AX: Motor Links/Rechtslauf Automatisierung IE



• — ○ • — ○ • — ○ •

Einleitung

Diese Übung implementiert eine Steuerung für einen Motor mit Links- und Rechtslauf unter Verwendung des Adapter-Konzepts (AX) in 4diac. Ziel der Übung ist es, zwei Ausgänge (Q5 und Q6) über separate Eingangsereignisse zu steuern (Setzen/Rücksetzen) und dabei einen gemeinsamen Statusausgang (Q56) zu aktivieren, sobald einer der beiden Motorausgänge aktiv ist. Zudem wird eine Verzögerung beim Umschaltvorgang über Taster I2 realisiert.

Verwendete Funktionsbausteine (FBs)

In dieser Übung werden verschiedene Standard-Bausteine aus der `logiBUS`-Bibliothek sowie logische Adapter-Bausteine verwendet.

Sub-Bausteine: Logik und IO

Hier werden die spezifischen Funktionsbausteine beschrieben, die für die Logik und die Hardware-Anbindung zuständig sind.

- **DigitalInput_CLK_I1 / I2 / I3**
  - **Typ**: `logiBUS::io::DI::logiBUS_IE`
  - **Beschreibung**: Dient der Erfassung von Taster-Eingaben.
  - **Parameter**:
    - `QI` = `TRUE`
    - `Input` = `Input_I1` (bzw. I2, I3)
    - `InputEvent` = `BUTTON_SINGLE_CLICK` (für I1), `BUTTON_PRESS_DOWN` (für I2, I3)
  - **Funktionsweise**: Wandelt Hardware-Signale in IEC 61499 Events um.
- **DigitalOutput_Q5 / Q6 / Q56**
  - **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
  - **Beschreibung**: Adapter-basierte Ausgänge zur Ansteuerung der Hardware (Motorrelais oder LEDs).
  - **Parameter**:
    - `QI` = `TRUE`
    - `Output` = `Output_Q5` (bzw. Q6, Q56)
  - **Funktionsweise**: Gibt den Status des verbundenen Adapters an die Hardware aus.
- **AX_SR_A / AX_SR_B**
  - **Typ**: `adapter::events::unidirectional::AX_SR`
  - **Beschreibung**: Speicherbaustein (Flip-Flop) basierend auf Adapter-Verbindungen.
  - **Funktionsweise**: Speichert den Zustand „Ein“ (Set) oder „Aus“ (Reset). `AX_SR_A` steuert den Pfad für Q5, `AX_SR_B` den Pfad für Q6.
    - Eingang S (Set): Aktiviert den Adapter-Ausgang Q.
    - Eingang R (Reset): Deaktiviert den Adapter-Ausgang Q.
- **AX_SPLIT_2_A / AX_SPLIT_2_B**
  - **Typ**: `adapter::events::unidirectional::AX_SPLIT_2`
  - **Beschreibung**: Signalverteiler für Adapter-Verbindungen.
  - **Funktionsweise**: Nimmt ein Adapter-Signal entgegen und stellt es an zwei Ausgängen (OUT1, OUT2) zur Verfügung. Dies ermöglicht, dass ein SR-Flip-Flop sowohl den direkten Motorausgang als auch das ODER-Gatter für die Statusanzeige speist.
- **AX_OR_2**
  - **Typ**: `adapter::booleanOperators::AX_OR_2`
  - **Beschreibung**: Logisches ODER für Adapter.
  - **Funktionsweise**: Der Ausgang OUT ist aktiv, wenn IN1 *oder* IN2 aktiv sind. Hier verwendet, um Q56 einzuschalten, wenn entweder Q5 oder Q6 läuft.
- **E_DELAY**
  - **Typ**: `iec61499::events::E_DELAY`
  - **Beschreibung**: Einschaltverzögerung.
  - **Parameter**:
    - `DT` = `T#50ms`
  - **Funktionsweise**: Verzögert das Event-Signal um 50 Millisekunden.

Programmablauf und Verbindungen

Das Netzwerk realisiert eine verriegelte Steuerung mit folgendem Ablauf:

1. **Motor Links (Q5) Starten:**
   - Das Event `BUTTON_SINGLE_CLICK` an **Input_I1** setzt den Baustein **AX_SR_A** (Eingang S).
   - Das Signal von `AX_SR_A` wird durch **AX_SPLIT_2_A** aufgeteilt:
     - Ein Pfad aktiviert direkt **DigitalOutput_Q5**.
     - Der zweite Pfad geht in das ODER-Gatter **AX_OR_2**, welches **DigitalOutput_Q56** (Betriebsanzeige) aktiviert.
2. **Umschalten / Stoppen Links (I2):**

- Das Event `BUTTON_PRESS_DOWN` an **Input_I2** hat zwei Funktionen:
  - Es setzt **AX_SR_A** zurück (Reset). Der Motor an Q5 stoppt sofort.
  - Gleichzeitig startet es den Timer **E_DELAY**.

3. **Motor Rechts (Q6) Starten (Verzögert):**
   - Nach Ablauf der Verzögerung (50ms) durch **E_DELAY** wird das Event `EO` ausgelöst.
   - Dieses Event setzt **AX_SR_B** (Eingang S).
   - Das Signal von `AX_SR_B` wird durch **AX_SPLIT_2_B** aufgeteilt:
     - Ein Pfad geht in das ODER-Gatter **AX_OR_2** und hält **DigitalOutput_Q56** aktiv.
     - Der zweite Pfad aktiviert **DigitalOutput_Q6**.
4. **Motor Rechts Stoppen (I3):**
   - Das Event `BUTTON_PRESS_DOWN` an **Input_I3** setzt **AX_SR_B** zurück (Reset). Der Motor an Q6 stoppt und Q56 erlischt (sofern Q5 nicht aktiv ist).

Zusammenfassend:

- **I1**: Startet Q5.
- **I2**: Stoppt Q5 und startet (nach 50ms) Q6.
- **I3**: Stoppt Q6.
- **Q56**: Leuchtet, wenn Q5 oder Q6 aktiv sind.

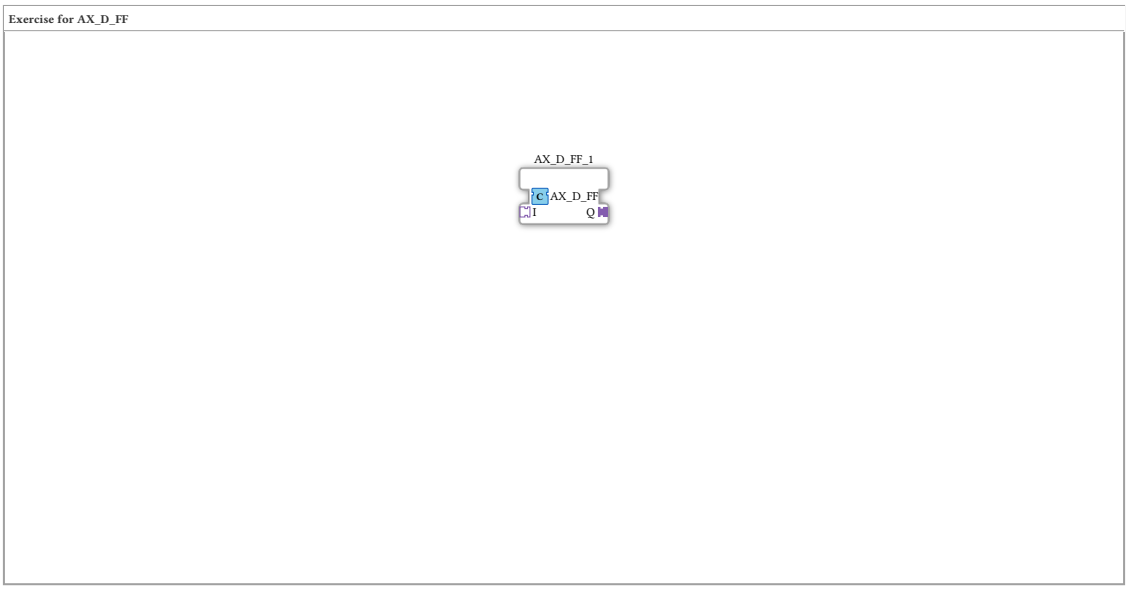
Zusammenfassung

Die Übung `Uebung_160b_AX` demonstriert die fortgeschrittene Nutzung von Adapter-Bausteinen zur Kapselung von Logik und Datenfluss. Durch den Einsatz von Splittern (`AX_SPLIT`) und logischen Operatoren (`AX_OR`) auf Adapter-Ebene wird der Schaltplan übersichtlich gehalten, während gleichzeitig komplexe Abhängigkeiten (gemeinsame Statusanzeige, Umschaltlogik) realisiert werden. Die Verzögerung (`E_DELAY`) sorgt für eine kurze Totzeit beim Umschalten der Drehrichtung über I2, was in realen Motoranwendungen wichtig für den Maschinenschutz ist.

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

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de
- 🌐 IEC 61499 Events – Der Puls der Automatisierung auf ms-muc-docs.de

2.1.2.454 Uebung_170_AX: Exercise for AX_D_FF



• – ◦ • – ◦ • – ◦ •

Einleitung

Die Sub-Applikation **Uebung_170_AX** ist eine grundlegende Übungseinheit, die speziell für den Funktionsbaustein `AX_D_FF` erstellt wurde. Ziel dieser Übung ist es, die Instanziierung und das Verhalten dieses spezifischen Adapter-Bausteins in einer IEC 61499 Umgebung zu demonstrieren.

Verwendete Funktionsbausteine (FBs)

In dieser Übung wird der folgende Hauptbaustein innerhalb des SubApp-Netzwerks verwendet:

Sub-Bausteine: AX_D_FF_1

Dieser Baustein ist die zentrale Komponente dieser Sub-Applikation.

- **Typ**: `adapter::events::unidirectional::AX_D_FF`
- **Verwendete interne FBs**:
  - Da es sich hierbei um die Instanziierung eines Bibliothekselements handelt, sind die internen FBs dieses Bausteins in dessen eigener Typ-Definition verborgen und nicht in dieser SubApp-Datei sichtbar.
- **Funktionsweise**:
  - Der Baustein `AX_D_FF` (D-Flip-Flop Adapter) dient vermutlich der Speicherung von Zuständen basierend auf Event-Triggern innerhalb einer unidirektionalen Adapter-Struktur. Er kapselt die Logik eines D-Flip-Flops für die Verwendung in Adapter-basierten Event-Ketten.

Programmablauf und Verbindungen

🌐 Netzwerkstruktur

Das Netzwerk dieser Sub-Applikation ist minimalistisch aufgebaut:

- Es enthält eine einzelne Instanz des Bausteins `AX_D_FF` (benannt als `AX_D_FF_1`).
- Positioniert bei den Koordinaten x=−1700, y=0.

Verbindungen

In der vorliegenden Definition sind **keine expliziten Event- oder Datenverbindungen** innerhalb dieser Sub-Applikation definiert.

- Dies deutet darauf hin, dass diese Übung entweder als Vorlage dient, in der der Lernende Verbindungen hinzufügen muss, oder dass der Baustein über Adapter-Schnittstellen (Plugs/Sockets) verfügt, die auf einer höheren Ebene verbunden werden.
- Die Übung konzentriert sich auf die Bereitstellung der `AX_D_FF` Instanz.

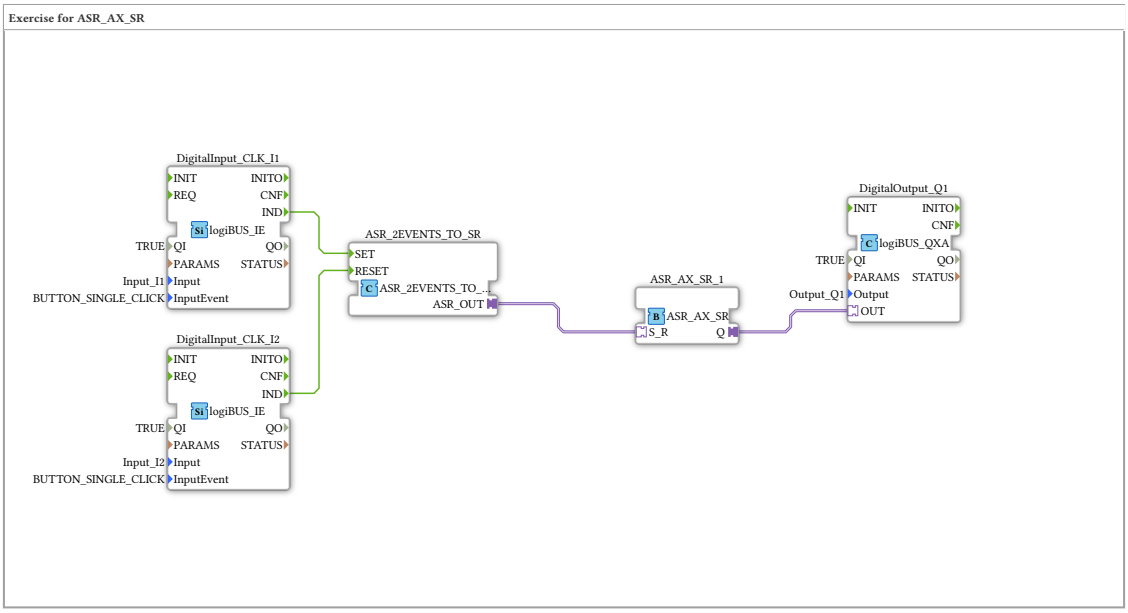
Lernziele & Hinweise

- **Schwierigkeitsgrad**: Einsteiger.
- **Vorkenntnisse**: Verständnis von Adaptern und D-Flip-Flops in IEC 61499.
- **Starten der Übung**: Platzieren Sie diese SubApp in einer Applikation und verbinden Sie die entsprechenden Adapter-Schnittstellen, um das Schaltverhalten des Flip-Flops zu beobachten.

Zusammenfassung

Die **Uebung_170_AX** stellt eine isolierte Umgebung für den `AX_D_FF` Baustein bereit. Sie dient als Container für dieses spezifische Adapter-Element, ohne dabei interne Verschaltungen vorzugeben, und bildet somit einen Baustein für komplexere Steuerungsaufgaben oder Testszenarien.

2.1.2.455 Uebung_171_ASR: Exercise for ASR_AX_SR



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung zeigt die Anwendung eines asynchronen Set-Reset-Flipflops (ASR) in der 4diac-IDE. Zwei Taster an den digitalen Eingängen I1 und I2 steuern das Setzen und Rücksetzen eines Speicherbausteins, dessen Ausgang einen digitalen Ausgang Q1 schaltet. Die Übung vermittelt grundlegende Konzepte der Ereignisverarbeitung und der Kopplung von Hardware-Eingängen mit einem RS-Speichermodul.

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: DigitalInput_CLK_I1 und DigitalInput_CLK_I2

- **Typ**: `logiBUS::io::DI::logiBUS_IE`
- **Verwendete interne FBs**: keine (Hardware-Konfigurationsbaustein)
  - **Parameter**:
    - `QI` = TRUE
    - `Input` = `Input_I1` bzw. `Input_I2`
    - `InputEvent` = `BUTTON_SINGLE_CLICK`
  - **Ereignisausgang**: `IND` (wird bei Betätigung des Tasters ausgelöst)
  - **Datenausgang**: keine
- **Funktionsweise**: Die Bausteine repräsentieren die digitalen Eingänge der logiBUS-Hardware. Sie detektieren einen einzelnen Tastendruck (Single Click) auf dem entsprechenden Eingangskanal und geben ein Ereignis (`IND`) aus.

Sub-Baustein: ASR_2EVENTS_TO_SR

- **Typ**: `adapter::conversion::unidirectional::ASR_2EVENTS_TO_SR`
- **Verwendete interne FBs**: keine (Konvertierungsbaustein)
  - **Parameter**: keine
  - **Ereigniseingänge**: `SET`, `RESET`
  - **Adapterausgang**: `ASR_OUT` (verbindet sich mit einem ASR-Adapter)
- **Funktionsweise**: Dieser Baustein wandelt zwei separate Ereignisse (SET und RESET) in eine Adapter-Schnittstelle um, die das Ansteuern eines ASR-Flipflops ermöglicht. Ein eingehendes SET-Ereignis setzt den Ausgangsadapter auf den Set-Zustand, ein RESET-Ereignis auf den Reset-Zustand.

Sub-Baustein: ASR_AX_SR_1

- **Typ**: `adapter::events::unidirectional::ASR_AX_SR`
- **Verwendete interne FBs**: keine (ASR-Speicherbaustein)
  - **Parameter**: keine

  - **Adaptereingang**: `S_R` (erhält SET/RESET-Signale vom Konverter)
  - **Datenausgang**: `Q` (boolescher Wert, Zustand des Flipflops)
- **Funktionsweise**: Der Baustein realisiert ein asynchrones Set-Reset-Flipflop. Der interne Zustand wird über den Adaptereingang `S_R` gesteuert: Ein Set-Signal aktiviert den Ausgang `Q` (TRUE), ein Reset-Signal deaktiviert ihn (FALSE). Der Ausgang bleibt bis zum nächsten Signal stabil.

Sub-Baustein: DigitalOutput_Q1

- **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
- **Verwendete interne FBs**: keine (Hardware-Konfigurationsbaustein)
  - **Parameter**:
    - `QI` = TRUE
    - `Output` = `Output_Q1`
  - **Daten eingang**: `OUT` (erhält den Schaltbefehl vom ASR)
  - **Ereignisausgang**: keine
- **Funktionsweise**: Der Baustein steuert den digitalen Ausgang Q1 der logiBUS-Hardware. Sobald am Dateneingang `OUT` ein TRUE-Signal anliegt, wird der angeschlossene Aktor (z. B. eine Lampe) eingeschaltet; bei FALSE wird er ausgeschaltet.

Programmablauf und Verbindungen

Der Ablauf wird durch die Ereignis- und Datenverbindungen im SubApp-Netzwerk bestimmt:

1. **Eingangsereignisse**:
   - Ein Tastendruck an `Input_I1` löst im Baustein `DigitalInput_CLK_I1` das Ereignis `IND` aus. Dieses wird an den Ereigniseingang `SET` des Konverters `ASR_2EVENTS_TO_SR` geleitet.
   - Ein Tastendruck an `Input_I2` löst im Baustein `DigitalInput_CLK_I2` das Ereignis `IND` aus. Dieses wird an den Ereigniseingang `RESET` des Konverters geleitet.
2. **Adapterverarbeitung**:
   - Der Konverter `ASR_2EVENTS_TO_SR` setzt den Ausgangsadapter `ASR_OUT` entsprechend dem letzten eingehenden Ereignis (SET oder RESET).
   - Der Adapterausgang ist mit dem Adaptereingang `S_R` des ASR-Bausteins `ASR_AX_SR_1` verbunden.
3. **Speicher und Ausgabe**:
   - Der ASR-Baustein reagiert auf das anliegende Adaptersignal und aktualisiert seinen Ausgang `Q`.
   - Der Ausgang `Q` ist mit dem Dateneingang `OUT` des digitalen Ausgangsbausteins `DigitalOutput_Q1` verbunden. Dadurch wird der physische Ausgang Q1 entsprechend ein- oder ausgeschaltet.

- **Lernziele**: Verständnis der Ereignissteuerung, Umgang mit Adapterbausteinen, einfache Speicherfunktion (RS-Flipflop).
- **Schwierigkeitsgrad**: Mittel
- **Benötigte Vorkenntnisse**: Grundlagen der 4diac-IDE, Umgang mit Ereignis- und Datenverbindungen, logiBUS-Konfiguration.
- **Ausführung**: Die Übung kann nach dem Laden und Kompilieren in der 4diac-Runtime ausgeführt werden. Die Eingangskanäle I1 und I2 müssen mit Tastern belegt sein; der Ausgang Q1 steuert einen Aktor (z. B. LED oder Relais).

Zusammenfassung

Die Übung `Uebung_171_ASR` demonstriert die Realisierung eines asynchronen RS-Speichers mit zwei Tastern als Eingänge und einem digitalen Ausgang. Durch die Kombination von Hardware-Konfigurationsbausteinen (logiBUS), einem Ereignis-zu-Adapter-Konverter und einem ASR-Speichermodul wird eine einfache, aber typische Steuerungsaufgabe abgebildet. Der Nutzer lernt, wie diskrete Ereignisse über Adapter an einen Speicherbaustein weitergegeben und schließlich auf einen physischen Ausgang geschaltet werden.

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

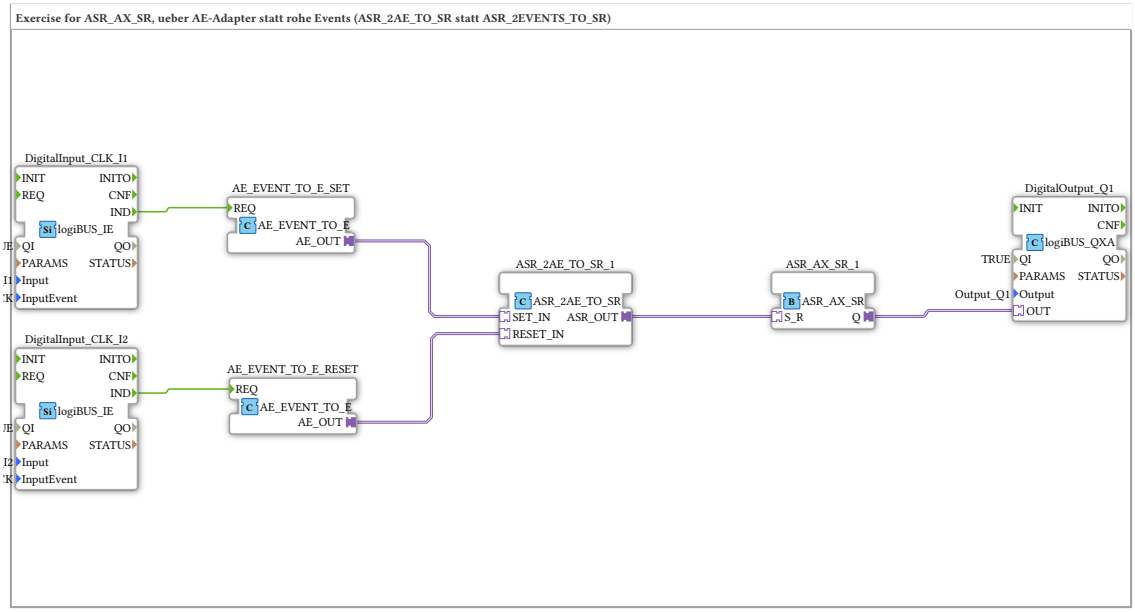
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.1.2.456 Uebung_171_AX

Hinweis

Diese Übung existiert in den aktuellen Trainingsquellen nicht mehr (entfernt, zusammengelegt oder umbenannt). Diese Seite bleibt nur als Platzhalter für bestehende Verweise erhalten. Die aktuellen Übungen stehen in der Übungsübersicht.

2.1.2.457 Uebung_171b_ASR: SET/RESET-Latch über AE-Adapter statt rohe Events



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung realisiert dieselbe Funktion wie `Uebung_171_ASR` (Taster `I1` = SET-Klick, Taster `I2` = RESET-Klick, Verriegelung über `ASR_AX_SR`), verdrahtet den Weg von den beiden Klick-Events zum Latch jedoch bewusst über **AE-Adapter-Sockets** statt über rohe EventConnections. Sie zeigt damit, warum es den AE-Adapter-Typ überhaupt gibt: ein `AE`-Plug ist ein vollwertiger Adapter, der sich wie `AX`/`ASR`/`ASRT` mit generischen Bausteinen (z. B. `AE_SPLIT_2`) weiterverarbeiten lässt – ein rohes Event kann das nicht, es ist nur über EventConnections verdrahtbar.

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_CLK_I1**, **DigitalInput_CLK_I2**: logiBUS Klick-Eingänge (Typ: `logiBUS::io::DI::logiBUS_IE`)
  - **Parameter**: `QI = TRUE`, `Input = Input_I1` bzw. `Input_I2`, `InputEvent = BUTTON_SINGLE_CLICK`
  - **Erklärung**: Melden einen einzelnen Tasterklick als Ereignis (`IND`) am jeweiligen physischen Eingang.
- **AE_EVENT_TO_E_SET**, **AE_EVENT_TO_E_RESET**: Ereignis-zu-Adapter-Brücke (Typ: `adapter::conversion::unidirectional::AE_EVENT_TO_E`)
  - **Parameter**: keine
  - **Erklärung**: Wandeln ein rohes Klick-Event (`REQ`) in einen getypten `AE`-Adapter-Ausgang (`AE_OUT`) um. Damit lässt sich das Ereignis wie ein Adapter weiterreichen statt nur als lose EventConnection.
- **ASR_2AE_TO_SR_1**: AE-zu-ASR-Konverter (Typ: `adapter::conversion::unidirectional::ASR_2AE_TO_SR`)
  - **Parameter**: keine
  - **Erklärung**: Nimmt zwei `AE`-Adapter-Eingänge (`SET_IN`, `RESET_IN`) entgegen und bündelt sie zu einem gemeinsamen `ASR`-Adapter-Ausgang (`ASR_OUT`).
- **ASR_AX_SR_1**: Set-Reset-Latch (Typ: `adapter::events::unidirectional::ASR_AX_SR`)
  - **Parameter**: keine
  - **Erklärung**: Ereignisgesteuerte bistabile Kippstufe für den gebündelten `S_R`-Eingang (ASR-Adapter): das Ereignis `SET` schaltet `Q` EIN, das Ereignis `RESET` schaltet `Q` AUS, das zuletzt eingetroffene Ereignis gewinnt.

- **DigitalOutput_Q1**: logiBUS Digitalausgang (Typ: `logiBUS::io::DQ::logiBUS_QXA`)
  - **Parameter**: `QI = TRUE`, `Output = Output_Q1`
  - **Erklärung**: Gibt den Latch-Zustand `Q` als physisches Ausgangssignal aus.

Sub-Bausteine: keine

Die Übung verwendet keine weiteren Unterbausteine, alle FBs sind direkt auf der obersten Ebene der SubApp angeordnet.

Programmablauf und Verbindungen

1. **Klick an I1 (SET)**: `DigitalInput_CLK_I1.IND` löst über eine EventConnection `AE_EVENT_TO_E_SET.REQ` aus. Dieser wandelt das rohe Event in einen `AE`-Adapter-Ausgang (`AE_OUT`) um.
2. **Klick an I2 (RESET)**: analog löst `DigitalInput_CLK_I2.IND` `AE_EVENT_TO_E_RESET.REQ` aus, dessen `AE_OUT` den Reset-Zweig bedient.
3. **Bündeln zu ASR**: `AE_EVENT_TO_E_SET.AE_OUT` geht über eine AdapterConnection auf `ASR_2AE_TO_SR_1.SET_IN`, `AE_EVENT_TO_E_RESET.AE_OUT` auf `ASR_2AE_TO_SR_1.RESET_IN`. Der Baustein fasst beide zu einem gemeinsamen `ASR_OUT` zusammen.
4. **Verriegeln**: `ASR_2AE_TO_SR_1.ASR_OUT` wird auf `ASR_AX_SR_1.S_R` geführt. Das Latch schaltet bei SET auf `Q = TRUE`, bei RESET auf `Q = FALSE`.
5. **Ausgabe**: `ASR_AX_SR_1.Q` treibt direkt `DigitalOutput_Q1.OUT`.

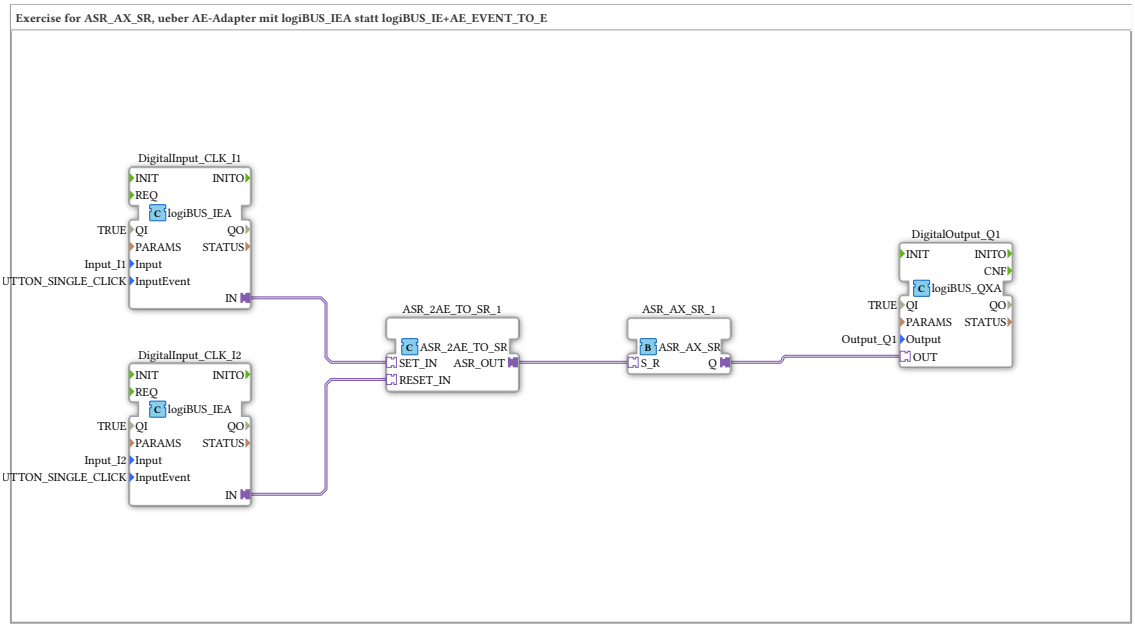
Zusammenfassung

Die Übung 171b demonstriert denselben SET/RESET-Latch wie Übung 171, verdrahtet aber bewusst über AE-Adapter-Sockets statt rohe Events: `AE_EVENT_TO_E` baut je Kanal die Brücke vom Klick-Event zum getypten `AE`-Plug, `ASR_2AE_TO_SR` bündelt beide zu einem `ASR`-Signal für das Latch `ASR_AX_SR`. Der Mehraufwand gegenüber der rohen Event-Variante lohnt sich, sobald das Signal selbst wie ein Adapter weiterverarbeitet werden soll (z. B. mit `AE_SPLIT_2` vervielfältigt) – etwas, das mit einem rohen Event nicht möglich wäre.

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

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

2.1.2.458 Uebung_171c_ASR: SET/RESET-Latch mit logiBUS_IEA (AE-Adapter bereits eingebaut)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung realisiert dieselbe Funktion wie `Uebung_171b_ASR` (Taster `I1` = SET-Klick, Taster `I2` = RESET-Klick, Verriegelung über `ASR_AX_SR`), spart sich aber die dort noch von Hand verdrahtete `AE_EVENT_TO_E`-Brücke pro Kanal: der Eingangsbaustein `logiBUS_IEA` liefert den `AE`-Adapter-Plug bereits eingebaut, genau wie `logiBUS_IX` ⟶ `logiBUS_IXA` für Pegelsignale, nur für Event-Eingänge.

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_CLK_I1**, **DigitalInput_CLK_I2**: logiBUS Klick-Eingänge mit eingebautem AE-Adapter (Typ: `logiBUS::io::DI::logiBUS_IEA`)
  - **Parameter**: `QI = TRUE`, `Input = Input_I1` bzw. `Input_I2`, `InputEvent = BUTTON_SINGLE_CLICK`
  - **Erklärung**: Kapseln intern einen `logiBUS_IE` und leiten dessen Klick-Event direkt auf einen eigenen `AE`-Adapter-Ausgang (`IN`) um – die Brücke, die in `Uebung_171b_ASR` noch als separater `AE_EVENT_TO_E`-Baustein nötig war, steckt hier bereits im Eingangsbaustein selbst.
- **ASR_2AE_TO_SR_1**: AE-zu-ASR-Konverter (Typ: `adapter::conversion::unidirectional::ASR_2AE_TO_SR`)
  - **Parameter**: keine
  - **Erklärung**: Nimmt zwei `AE`-Adapter-Eingänge (`SET_IN`, `RESET_IN`) entgegen und bündelt sie zu einem gemeinsamen `ASR`-Adapter-Ausgang (`ASR_OUT`).
- **ASR_AX_SR_1**: Set-Reset-Latch (Typ: `adapter::events::unidirectional::ASR_AX_SR`)
  - **Parameter**: keine
  - **Erklärung**: Verriegelt den gebündelten `S_R`-Eingang: `S` schaltet `Q` EIN, `R` schaltet `Q` AUS. Es gibt keine Dominanz, das zuletzt eingetroffene Ereignis bestimmt den Zustand („Last Event Wins“).
- **DigitalOutput_Q1**: logiBUS Digitalausgang (Typ: `logiBUS::io::DQ::logiBUS_QXA`)
  - **Parameter**: `QI = TRUE`, `Output = Output_Q1`
  - **Erklärung**: Gibt den Latch-Zustand `Q` als physisches Ausgangssignal aus.

Sub-Bausteine: keine

Die Übung verwendet keine weiteren Unterbausteine, alle FBs sind direkt auf der obersten Ebene der SubApp angeordnet.

Programmablauf und Verbindungen

1. **Klick an I1 (SET)**: `DigitalInput_CLK_I1` (`logiBUS_IEA`) erkennt den Klick intern und stellt ihn direkt als `AE`-Adapter-Ausgang (`IN`) bereit – ohne separaten Zwischenbaustein.
2. **Klick an I2 (RESET)**: analog liefert `DigitalInput_CLK_I2.IN` den Reset-Zweig als `AE`-Adapter.
3. **Bündeln zu ASR**: `DigitalInput_CLK_I1.IN` geht direkt auf `ASR_2AE_TO_SR_1.SET_IN`, `DigitalInput_CLK_I2.IN` auf `ASR_2AE_TO_SR_1.RESET_IN`. Der Baustein fasst beide zu `ASR_OUT` zusammen.
4. **Verriegeln**: `ASR_2AE_TO_SR_1.ASR_OUT` wird auf `ASR_AX_SR_1.S_R` geführt. Das Latch schaltet bei SET auf `Q = TRUE`, bei RESET auf `Q = FALSE`.
5. **Ausgabe**: `ASR_AX_SR_1.Q` treibt direkt `DigitalOutput_Q1.OUT`.

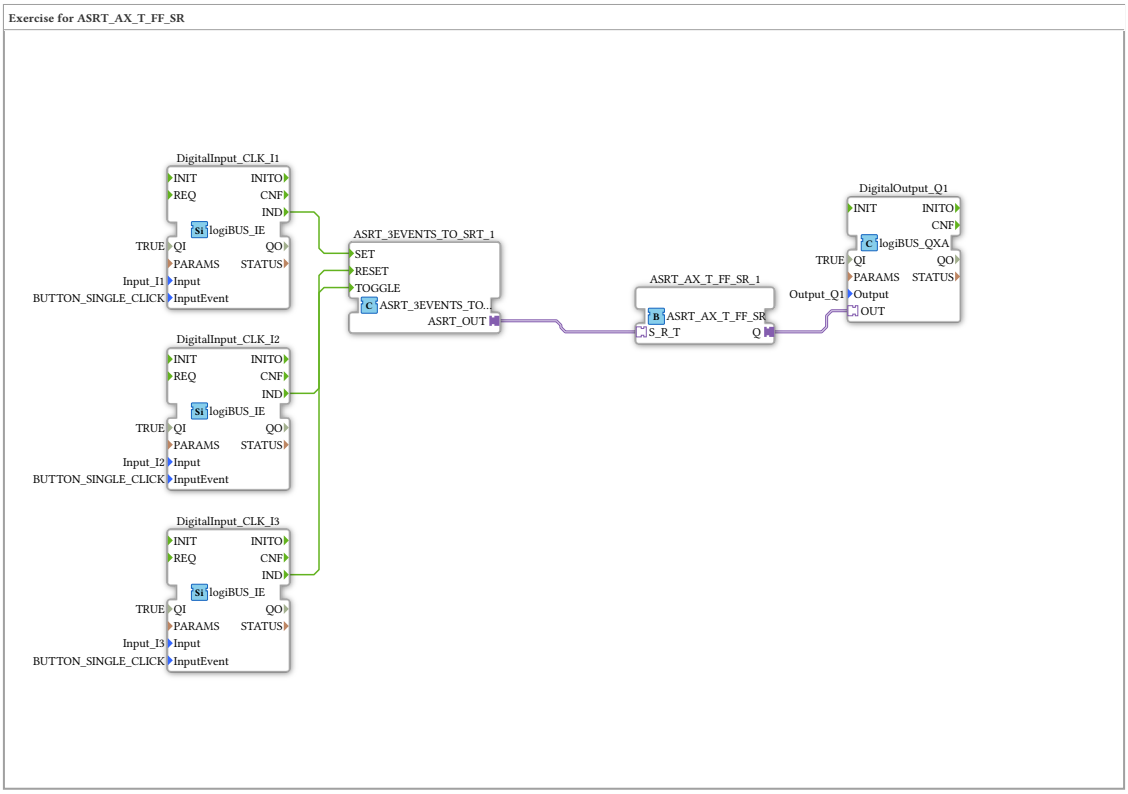
Zusammenfassung

Übung 171c zeigt exakt dasselbe Latch-Verhalten wie Übung 171 und 171b, kommt aber mit den wenigsten Bausteinen aus: `logiBUS_IEA` ersetzt `logiBUS_IE` und liefert den `AE`-Adapter-Plug bereits eingebaut, sodass die in 171b noch nötigen `AE_EVENT_TO_E`-Bausteine entfallen. Im Vergleich der drei SubApps im 4diac-Editor zeigt sich derselbe funktionale Kern (Klick I1 ⟶ `Q1` EIN, Klick I2 ⟶ `Q1` AUS) mit schrittweise kompakterer Verdrahtung.

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

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

2.1.2.459 Uebung_172_ASRT: Exercise for ASRT_AX_T_FF_SR



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung zeigt die Anwendung eines SR+Toggle-Flipflops (ASRT) in der 4diac-IDE. Drei Taster an den digitalen Eingängen I1, I2 und I3 steuern das Setzen, Rücksetzen und Umschalten eines Speicherbausteins, dessen Ausgang einen digitalen Ausgang Q1 schaltet. Die Übung ist der direkte Nachfolger von `Uebung_171_ASR` und erweitert das reine SR-Flipflop um einen dritten, umschaltenden Eingang.

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: DigitalInput_CLK_I1, DigitalInput_CLK_I2 und DigitalInput_CLK_I3

- **Typ**: `logiBUS::io::DI::logiBUS_IE`
- **Verwendete interne FBs**: keine (Hardware-Konfigurationsbaustein)
  - **Parameter**:
    - `QI` = TRUE
    - `Input` = `Input_I1`, `Input_I2` bzw. `Input_I3`
    - `InputEvent` = `BUTTON_SINGLE_CLICK`
  - **Ereignisausgang**: `IND` (wird bei Betätigung des Tasters ausgelöst)
  - **Datenausgang**: keine
- **Funktionsweise**: Die Bausteine repräsentieren die digitalen Eingänge der logiBUS-Hardware. Sie detektieren einen einzelnen Tastendruck (Single Click) auf dem entsprechenden Eingangskanal und geben ein Ereignis (`IND`) aus.

Sub-Baustein: ASRT_3EVENTS_TO_SRT_1

- **Typ**: `adapter::conversion::unidirectional::ASRT_3EVENTS_TO_SRT`
- **Verwendete interne FBs**: keine (Konvertierungsbaustein)
  - **Parameter**: keine
  - **Ereigniseingänge**: `SET`, `RESET`, `TOGGLE`
  - **Adapterausgang**: `ASRT_OUT` (verbindet sich mit einem ASRT-Adapter)
- **Funktionsweise**: Dieser Baustein wandelt drei separate Ereignisse (SET, RESET und TOGGLE) in eine Adapter-Schnittstelle um, die das Ansteuern eines ASRT-Flipflops ermöglicht. Ein eingehendes SET-

Ereignis setzt den Ausgangsadapter auf den Set-Zustand, ein RESET-Ereignis auf den Reset-Zustand, ein TOGGLE-Ereignis kehrt den aktuellen Zustand um.

Sub-Baustein: ASRT_AX_T_FF_SR_1

- **Typ**: `adapter::events::unidirectional::ASRT_AX_T_FF_SR`
- **Verwendete interne FBs**: keine (ASRT-Speicherbaustein)
  - **Parameter**: keine
  - **Adaptereingang**: `S_R_T` (erhält SET/RESET/TOGGLE-Signale vom Konverter)
  - **Datenausgang**: `Q` (boolescher Wert, Zustand des Flipflops)
- **Funktionsweise**: Der Baustein realisiert ein SR+Toggle-Flipflop. Der interne Zustand wird über den Adaptereingang `S_R_T` gesteuert: Ein Set-Signal aktiviert den Ausgang `Q` (TRUE), ein Reset-Signal deaktiviert ihn (FALSE), ein Toggle-Signal kehrt den aktuellen Zustand um. Der Ausgang bleibt bis zum nächsten Signal stabil.

Sub-Baustein: DigitalOutput_Q1

- **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
- **Verwendete interne FBs**: keine (Hardware-Konfigurationsbaustein)
  - **Parameter**:
    - `QI` = TRUE
    - `Output` = `Output_Q1`
  - **Daten eingang**: `OUT` (erhält den Schaltbefehl vom ASRT)
  - **Ereignisausgang**: keine
- **Funktionsweise**: Der Baustein steuert den digitalen Ausgang Q1 der logiBUS-Hardware. Sobald am Dateneingang `OUT` ein TRUE-Signal anliegt, wird der angeschlossene Aktor (z. B. eine Lampe) eingeschaltet; bei FALSE wird er ausgeschaltet.

Programmablauf und Verbindungen

Der Ablauf wird durch die Ereignis- und Adapterverbindungen im SubApp-Netzwerk bestimmt:

1. **Eingangsereignisse**:
   - Ein Tastendruck an `Input_I1` löst im Baustein `DigitalInput_CLK_I1` das Ereignis `IND` aus. Dieses wird an den Ereigniseingang `SET` des Konverters `ASRT_3EVENTS_TO_SRT_1` geleitet.
   - Ein Tastendruck an `Input_I2` löst im Baustein `DigitalInput_CLK_I2` das Ereignis `IND` aus. Dieses wird an den Ereigniseingang `RESET` des Konverters geleitet.
   - Ein Tastendruck an `Input_I3` löst im Baustein `DigitalInput_CLK_I3` das Ereignis `IND` aus. Dieses wird an den Ereigniseingang `TOGGLE` des Konverters geleitet.
2. **Adapterverarbeitung**:
   - Der Konverter `ASRT_3EVENTS_TO_SRT_1` setzt den Ausgangsadapter `ASRT_OUT` entsprechend dem letzten eingehenden Ereignis (SET, RESET oder TOGGLE).
   - Der Adapterausgang ist mit dem Adaptereingang `S_R_T` des ASRT-Bausteins `ASRT_AX_T_FF_SR_1` verbunden.
3. **Speicher und Ausgabe**:
   - Der ASRT-Baustein reagiert auf das anliegende Adaptersignal und aktualisiert seinen Ausgang `Q`.
   - Der Ausgang `Q` ist mit dem Dateneingang `OUT` des digitalen Ausgangsbausteins `DigitalOutput_Q1` verbunden. Dadurch wird der physische Ausgang Q1 entsprechend ein- oder ausgeschaltet.

- **Lernziele**: Verständnis der Ereignissteuerung mit drei unabhängigen Eingängen, Umgang mit Adapterbausteinen, kombinierte SR+Toggle-Speicherfunktion, Abgrenzung zum reinen SR-Flipflop aus `Uebung_171_ASR`.
- **Schwierigkeitsgrad**: Mittel
- **Benötigte Vorkenntnisse**: Grundlagen der 4diac-IDE, Umgang mit Ereignis- und Datenverbindungen, logiBUS-Konfiguration, Kenntnis der Übung `Uebung_171_ASR`.
- **Ausführung**: Die Übung kann nach dem Laden und Kompilieren in der 4diac-Runtime ausgeführt werden. Die Eingangskanäle I1, I2 und I3 müssen mit Tastern belegt sein; der Ausgang Q1 steuert einen Aktor (z. B. LED oder Relais).

Zusammenfassung

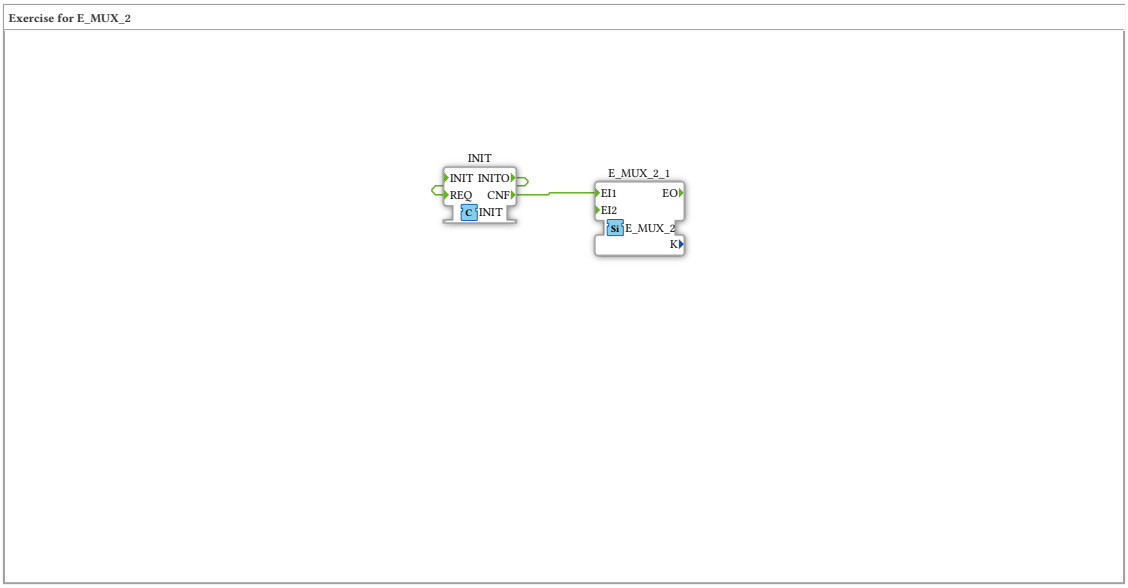
Die Übung `Uebung_172_ASRT` demonstriert die Realisierung eines SR+Toggle-Speichers mit drei Tastern als Eingänge und einem digitalen Ausgang. Sie ist der direkte Analog zu `Uebung_171_ASR`, ersetzt aber den ASR-Adapter durch den neuen ASRT-Adapter, der zusätzlich zu Set/Reset auch ein Toggle-Ereignis unterstützt. Der Nutzer lernt, wie drei diskrete Ereignisse über Adapter an einen kombinierten Speicherbaustein weitergegeben und schließlich auf einen physischen Ausgang geschaltet werden.

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

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

2.1.2.460 Uebung_172_AX: Exercise for E_MUX_2

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_172_AX (Exercise for E_MUX_2).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Exercise for E_MUX_2**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_172_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **E_MUX_2_1**: Instanz des Typs iec61499::events::E_MUX_2.
- **INIT**: Instanz des Typs iec61131::booleanOperators::INIT.

Verbindungen und Schnittstellen

Ereignisverbindungen:

- INIT.INITO -> INIT.REQ
- INIT.CNF -> E_MUX_2_1.EI1

Hinweise aus dem Modell

TODO

Programmablauf und Funktionsweise

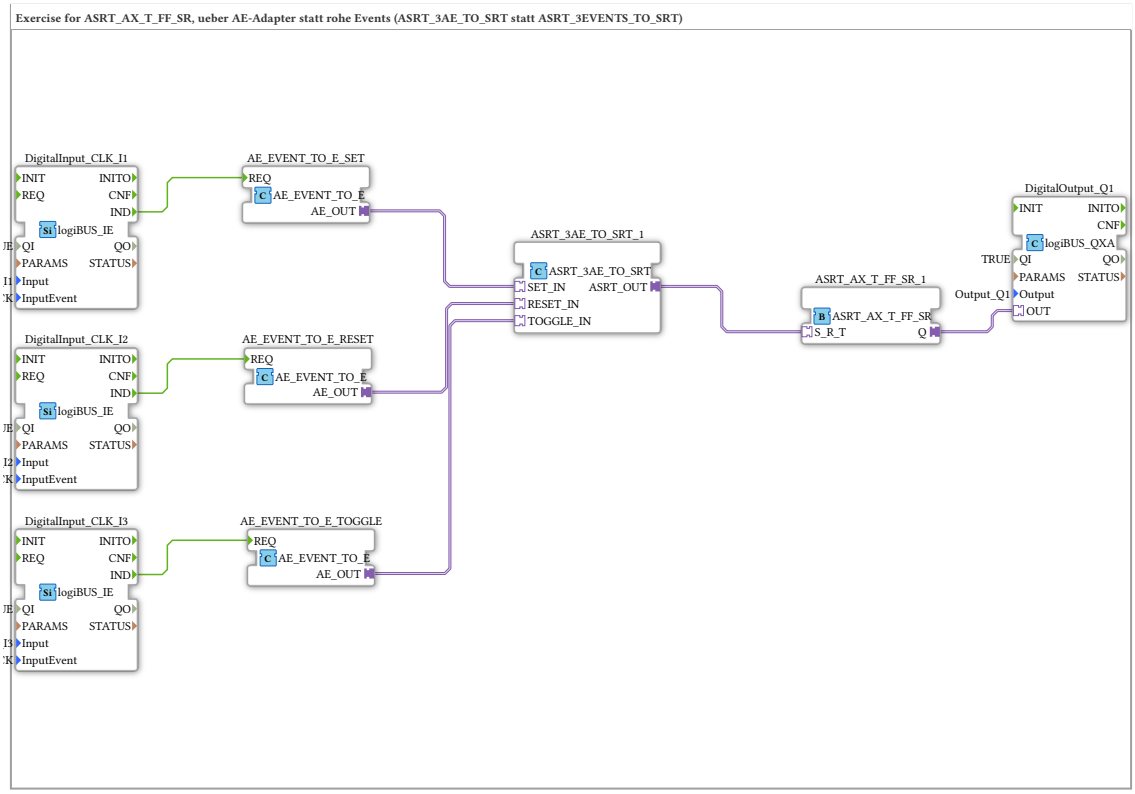
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.

2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_172_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.461 Uebung_172b_ASRT: SET/RESET/TOGGLE-Latch über AE-Adapter statt rohe Events



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung realisiert dieselbe Funktion wie `Uebung_172_ASRT` (Taster `I1` = SET-Klick, `I2` = RESET-Klick, `I3` = TOGGLE-Klick, Verriegelung über `ASRT_AX_T_FF_SR`), verdrahtet den Weg von den drei Klick-Events zum Latch jedoch über **AE-Adapter-Sockets** statt über rohe EventConnections. Wie bei `Uebung_171b_ASR` zeigt sich hier derselbe Grund: ein `AE`-Plug ist ein vollwertiger Adapter (weiterverarbeitbar z. B. mit `AE_SPLIT_2`), ein rohes Event dagegen nur über EventConnections verdrahtbar.

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_CLK_I1**, **DigitalInput_CLK_I2**, **DigitalInput_CLK_I3**: logiBUS Klick-Eingänge (Typ: `logiBUS::io::DI::logiBUS_IE`)
  - **Parameter**: `QI = TRUE`, `Input = Input_I1/Input_I2/Input_I3`, `InputEvent = BUTTON_SINGLE_CLICK`
  - **Erklärung**: Melden je einen einzelnen Tasterklick als Ereignis (`IND`) am jeweiligen physischen Eingang.
- **AE_EVENT_TO_E_SET**, **AE_EVENT_TO_E_RESET**, **AE_EVENT_TO_E_TOGGLE**: Ereignis-zu-Adapter-Brücke (Typ: `adapter::conversion::unidirectional::AE_EVENT_TO_E`)
  - **Parameter**: keine
  - **Erklärung**: Wandeln je ein rohes Klick-Event (`REQ`) in einen getypten `AE`-Adapter-Ausgang (`AE_OUT`) um.

- **ASRT_3AE_TO_SRT_1**: AE-zu-ASRT-Konverter
  (Typ: `adapter::conversion::unidirectional::ASRT_3AE_TO_SRT`)
  - **Parameter**: keine
  - **Erklärung**: Nimmt drei AE-Adapter-Eingänge (`SET_IN`, `RESET_IN`, `TOGGLE_IN`) entgegen und bündelt sie zu einem gemeinsamen ASRT-Adapter-Ausgang (`ASRT_OUT`).
- **ASRT_AX_T_FF_SR_1**: Set-Reset-Toggle-Latch
  (Typ: `adapter::events::unidirectional::ASRT_AX_T_FF_SR`)
  - **Parameter**: keine
  - **Erklärung**: Verriegelt den gebündelten `S_R_T`-Eingang: SET schaltet `Q` EIN, RESET schaltet `Q` AUS, TOGGLE kehrt den aktuellen Zustand um.
- **DigitalOutput_Q1**: logiBUS Digitalausgang (Typ: `logiBUS::io::DQ::logiBUS_QXA`)
  - **Parameter**: `QI = TRUE`, `Output = Output_Q1`
  - **Erklärung**: Gibt den Latch-Zustand `Q` als physisches Ausgangssignal aus.

Sub-Bausteine: keine

Die Übung verwendet keine weiteren Unterbausteine, alle FBs sind direkt auf der obersten Ebene der SubApp angeordnet.

Programmablauf und Verbindungen

1. **Klick an I1 (SET)**: `DigitalInput_CLK_I1.IND` löst `AE_EVENT_TO_E_SET.REQ` aus, der das Event in einen AE-Adapter-Ausgang wandelt.
2. **Klick an I2 (RESET)**: analog löst `DigitalInput_CLK_I2.IND` `AE_EVENT_TO_E_RESET.REQ` aus.
3. **Klick an I3 (TOGGLE)**: analog löst `DigitalInput_CLK_I3.IND` `AE_EVENT_TO_E_TOGGLE.REQ` aus.
4. **Bündeln zu ASRT**: die drei `AE_OUT`-Ausgänge gehen auf `ASRT_3AE_TO_SRT_1.SET_IN`/`RESET_IN`/`TOGGLE_IN`. Der Baustein fasst sie zu `ASRT_OUT` zusammen.
5. **Verriegeln**: `ASRT_3AE_TO_SRT_1.ASRT_OUT` wird auf `ASRT_AX_T_FF_SR_1.S_R_T` geführt. SET → `Q = TRUE`, RESET → `Q = FALSE`, TOGGLE → `Q` wechselt den Zustand.
6. **Ausgabe**: `ASRT_AX_T_FF_SR_1.Q` treibt direkt `DigitalOutput_Q1.OUT`.

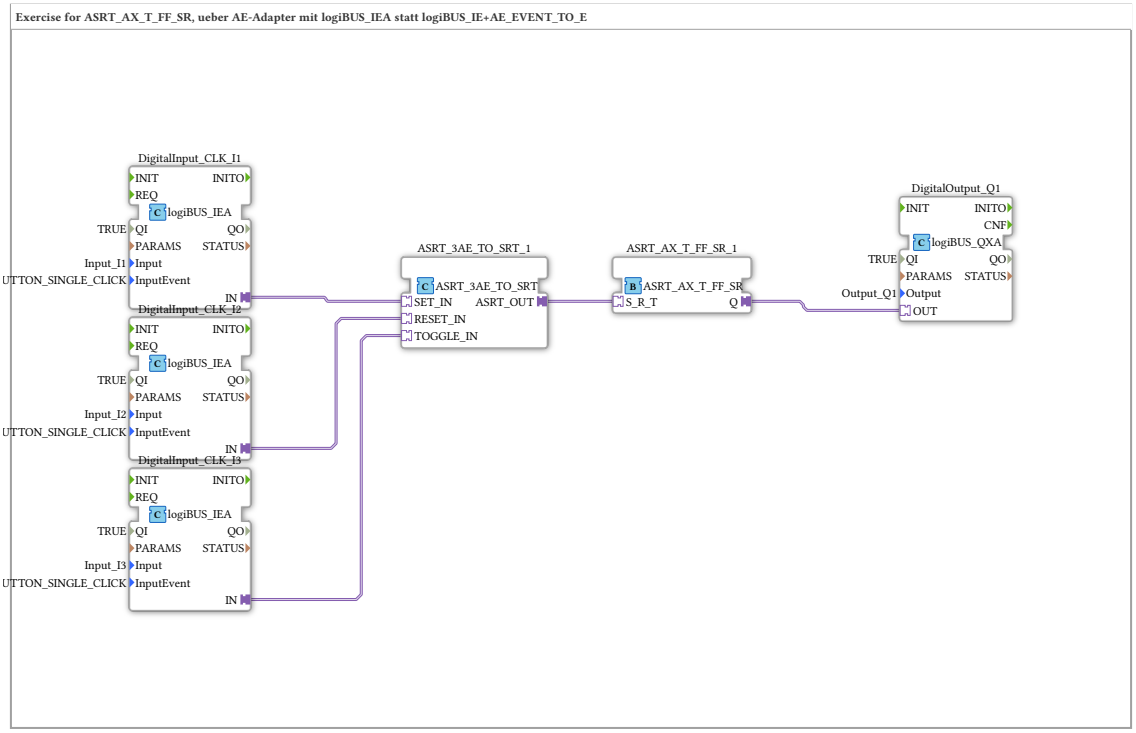
Zusammenfassung

Die Übung 172b demonstriert denselben SET/RESET/TOGGLE-Latch wie Übung 172, verdrahtet aber bewusst über AE-Adapter-Sockets statt rohe Events: je ein `AE_EVENT_TO_E` baut die Brücke vom Klick-Event zum getypten AE-Plug, `ASRT_3AE_TO_SRT` bündelt alle drei zu einem ASRT-Signal für das Latch `ASRT_AX_T_FF_SR`. Der Mehraufwand gegenüber der rohen Event-Variante lohnt sich, sobald das Signal selbst wie ein Adapter weiterverarbeitet werden soll.

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

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

2.1.2.462 Uebung_172c_ASRT: SET/RESET/TOGGLE-Latch mit logiBUS_IEA (AE-Adapter bereits eingebaut)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung realisiert dieselbe Funktion wie `Uebung_172b_ASRT` (Taster `I1` = SET-Klick, `I2` = RESET-Klick, `I3` = TOGGLE-Klick, Verriegelung über `ASRT_AX_T_FF_SR`), spart sich aber die dort noch von Hand verdrahtete `AE_EVENT_TO_E`-Brücke pro Kanal: der Eingangsbaustein `logiBUS_IEA` liefert den `AE`-Adapter-Plug bereits eingebaut, genau wie `logiBUS_IX` ⟶ `logiBUS_IXA` für Pegelsignale, nur für Event-Eingänge.

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_CLK_I1**, **DigitalInput_CLK_I2**, **DigitalInput_CLK_I3**: logiBUS Klick-Eingänge mit eingebautem AE-Adapter (Typ: `logiBUS::io::DI::logiBUS_IEA`)
  - **Parameter**: `QI = TRUE`, `Input = Input_I1/Input_I2/Input_I3`, `InputEvent = BUTTON_SINGLE_CLICK`
  - **Erklärung**: Kapseln intern einen `logiBUS_IE` und leiten dessen Klick-Event direkt auf einen eigenen `AE`-Adapter-Ausgang (`IN`) um – ohne separaten `AE_EVENT_TO_E`-Baustein wie in `Uebung_172b_ASRT`.
- **ASRT_3AE_TO_SRT_1**: AE-zu-ASRT-Konverter (Typ: `adapter::conversion::unidirectional::ASRT_3AE_TO_SRT`)
  - **Parameter**: keine
  - **Erklärung**: Nimmt drei `AE`-Adapter-Eingänge (`SET_IN`, `RESET_IN`, `TOGGLE_IN`) entgegen und bündelt sie zu einem gemeinsamen `ASRT`-Adapter-Ausgang (`ASRT_OUT`).
- **ASRT_AX_T_FF_SR_1**: Set-Reset-Toggle-Latch (Typ: `adapter::events::unidirectional::ASRT_AX_T_FF_SR`)
  - **Parameter**: keine
  - **Erklärung**: Verriegelt den gebündelten `S_R_T`-Eingang: SET schaltet `Q` EIN, RESET schaltet `Q` AUS, TOGGLE kehrt den aktuellen Zustand um.
- **DigitalOutput_Q1**: logiBUS Digitalausgang (Typ: `logiBUS::io::DQ::logiBUS_QXA`)
  - **Parameter**: `QI = TRUE`, `Output = Output_Q1`
  - **Erklärung**: Gibt den Latch-Zustand `Q` als physisches Ausgangssignal aus.

Sub-Bausteine: keine

Die Übung verwendet keine weiteren Unterbausteine, alle FBs sind direkt auf der obersten Ebene der SubApp angeordnet.

Programmablauf und Verbindungen

1. **Klick an I1 (SET)**: `DigitalInput_CLK_I1` (`logiBUS_IEA`) erkennt den Klick intern und stellt ihn direkt als `AE`-Adapter-Ausgang (`IN`) bereit.
2. **Klick an I2 (RESET)**: analog liefert `DigitalInput_CLK_I2.IN` den Reset-Zweig als `AE`-Adapter.
3. **Klick an I3 (TOGGLE)**: analog liefert `DigitalInput_CLK_I3.IN` den Toggle-Zweig als `AE`-Adapter.
4. **Bündeln zu ASRT**: die drei `IN`-Ausgänge gehen direkt auf `ASRT_3AE_TO_SRT_1.SET_IN/RESET_IN/TOGGLE_IN` – ohne Zwischenbaustein. Der Baustein fasst sie zu `ASRT_OUT` zusammen.
5. **Verriegeln**: `ASRT_3AE_TO_SRT_1.ASRT_OUT` wird auf `ASRT_AX_T_FF_SR_1.S_R_T` geführt. SET → `Q = TRUE`, RESET → `Q = FALSE`, TOGGLE → `Q` wechselt den Zustand.
6. **Ausgabe**: `ASRT_AX_T_FF_SR_1.Q` treibt direkt `DigitalOutput_Q1.OUT`.

Zusammenfassung

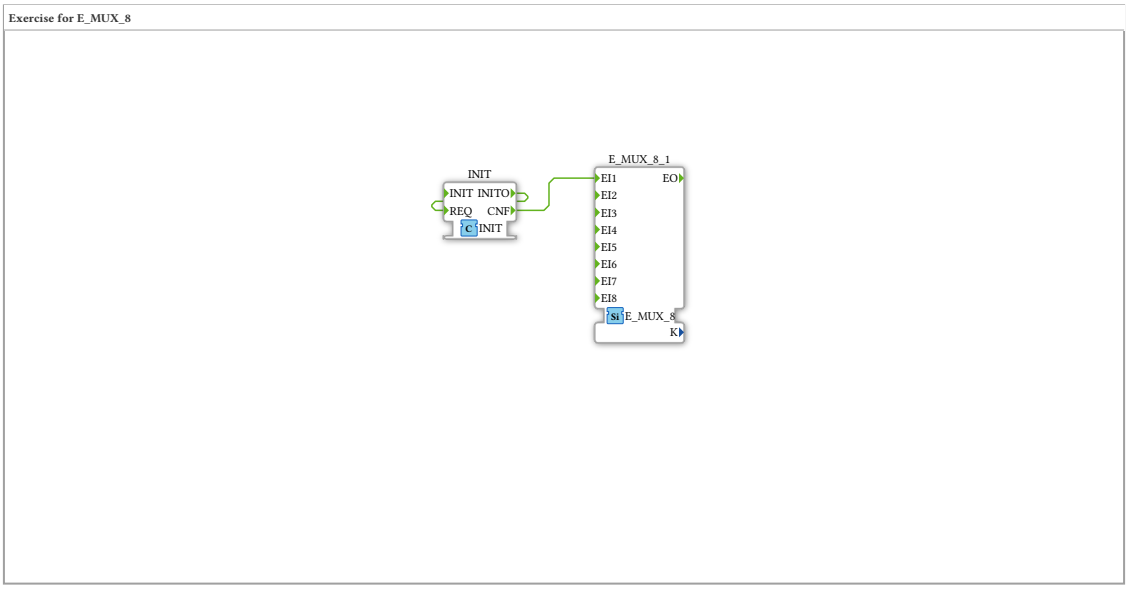
Übung 172c zeigt exakt dasselbe Latch-Verhalten wie Übung 172 und 172b, kommt aber mit den wenigsten Bausteinen aus: `logiBUS_IEA` ersetzt `logiBUS_IE` und liefert den `AE`-Adapter-Plug bereits eingebaut, sodass die in 172b noch nötigen drei `AE_EVENT_TO_E`-Bausteine vollständig entfallen. Im Vergleich aller drei SubApps im 4diac-Editor zeigt sich derselbe funktionale Kern mit schrittweise kompakterer Verdrahtung.

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

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

2.1.2.463 Uebung_173_AX: Exercise for E_MUX_8

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_173_AX (Exercise for E_MUX_8).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Exercise for E_MUX_8**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_173_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **E_MUX_8_1**: Instanz des Typs iec61499::events::E_MUX_8.
- **INIT**: Instanz des Typs iec61131::booleanOperators::INIT.

Verbindungen und Schnittstellen

Ereignisverbindungen:

- INIT.INITO -> INIT.REQ

- INIT.CNF -> E_MUX_8_1.EI1

Hinweise aus dem Modell

TODO

Programmablauf und Funktionsweise

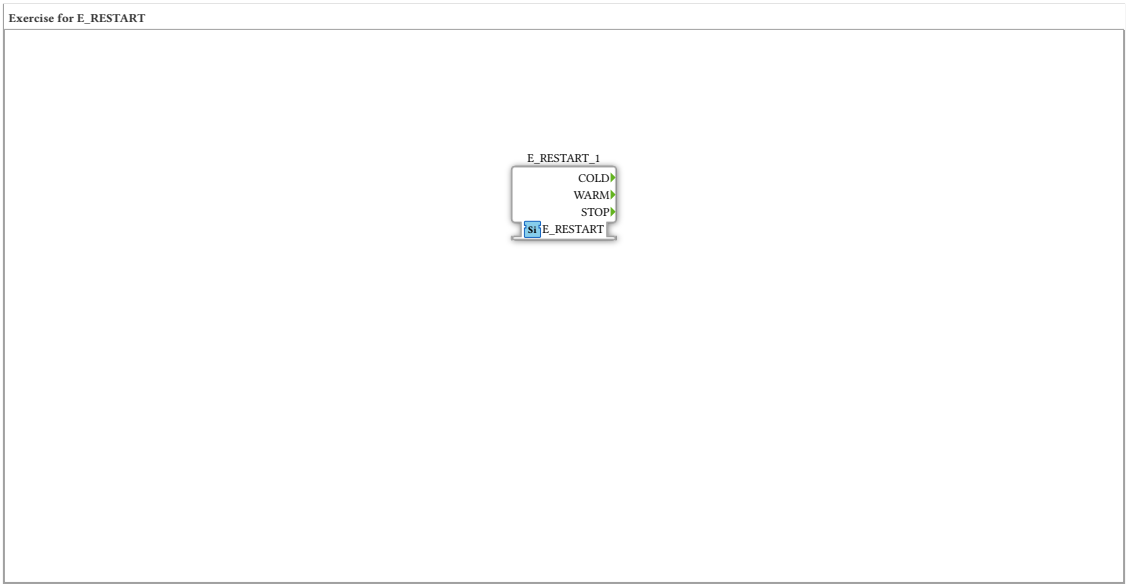
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_173_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.464 Uebung_174_AX: Exercise for E_RESTART

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_174_AX (Exercise for E_RESTART).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Exercise for E_RESTART**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_174_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **E_RESTART_1**: Instanz des Typs iec61499::events::E_RESTART.

Hinweise aus dem Modell

TODO

Programmablauf und Funktionsweise

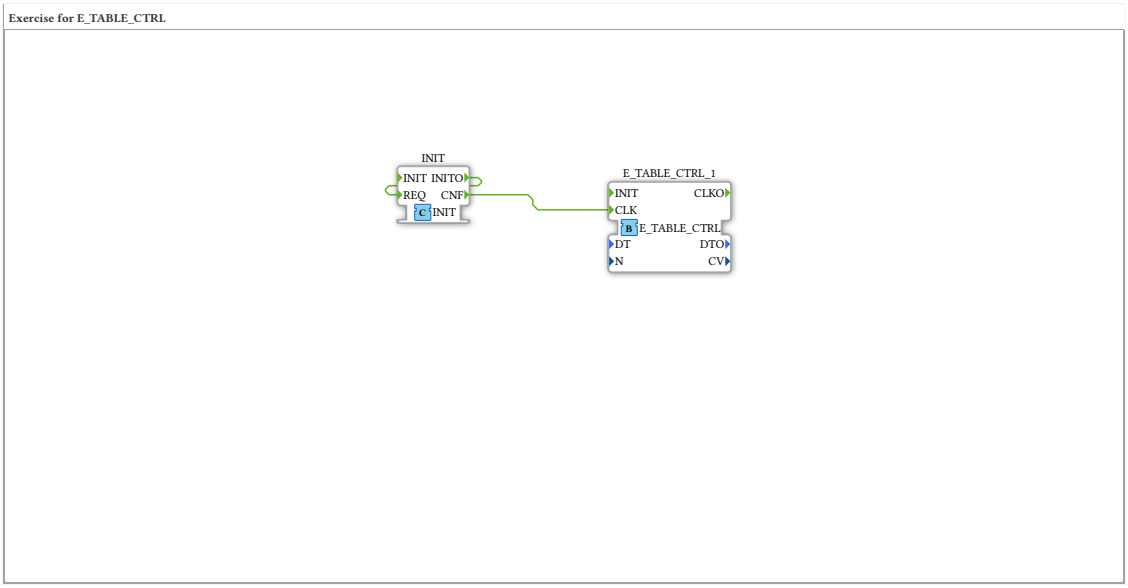
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Signalweiterleitung**: Die Bausteine verarbeiten Daten- und Steuersignale gemäß ihrer Konfiguration.

Zusammenfassung

Die Übung Uebung_174_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.465 Uebung_175_AX: Exercise for E_TABLE_CTRL

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_175_AX (Exercise for E_TABLE_CTRL).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Exercise for E_TABLE_CTRL**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_175_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **E_TABLE_CTRL_1**: Instanz des Typs iec61499::events::E_TABLE_CTRL.
- **INIT**: Instanz des Typs iec61131::booleanOperators::INIT.

Verbindungen und Schnittstellen

Ereignisverbindungen:

- INIT.INITO -> INIT.REQ
- INIT.CNF -> E_TABLE_CTRL_1.CLK

Hinweise aus dem Modell

TODO

Programmablauf und Funktionsweise

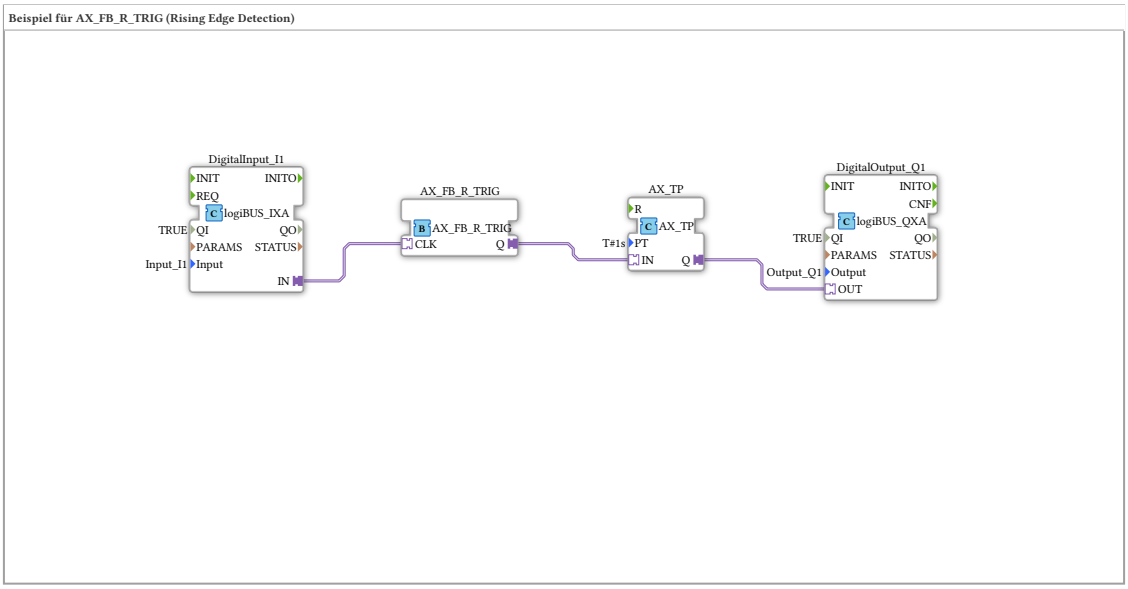
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.

3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_175_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.466 Uebung_177_AX: Beispiel für AX_FB_R_TRIG (Rising Edge Detection)



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung demonstriert die Verwendung der **Flankenerkennung (Rising Edge Detection)** in Kombination mit einem **Impulstimer**. Ziel ist es, ein Signal am physikalischen Eingang nur im Moment des Einschaltens (Wechsel von 0 auf 1) auszuwerten und daraufhin einen Ausgang für eine definierte Zeitdauer zu aktivieren.

Der Fokus liegt auf dem Baustein `AX_FB_R_TRIG`, der eine steigende Flanke erkennt.

Verwendete Funktionsbausteine (FBs)

In dieser Sub-Application kommen Hardware-Treiberbausteine sowie Logik- und Zeitbausteine zum Einsatz.

Sub-Bausteine

1. Hardware-Eingang

- **Name**: `DigitalInput_I1`
- **Typ**: `logiBUS::io::DI::logiBUS_IXA`
- **Parameter**:
  - `Input` = `Input_I1` (Verweis auf den physikalischen Eingang I1)
- **Funktionsweise**: Stellt den Zustand des digitalen Eingangs für die Logik bereit.

2. Flankenerkennung (Rising Trigger)

- **Name**: `AX_FB_R_TRIG`
- **Typ**: `adapter::iec61131::edgeDetection::AX_FB_R_TRIG`
- **Funktionsweise**: Dieser Baustein überwacht das Eingangssignal. Er gibt nur dann ein Signal am Ausgang `Q` aus, wenn das Eingangssignal `CLK` von `FALSE` (0) auf `TRUE` (1) wechselt (steigende Flanke). Dauersignale werden ignoriert.

3. Impuls-Timer (Pulse Timer)

- **Name**: `AX_TP`

- **Typ**: `adapter::events::unidirectional::timers::AX_TP`
- **Parameter**:
  - `PT` = `T#1s` (Process Time: 1 Sekunde)
- **Funktionsweise**: Erzeugt einen Impuls am Ausgang `Q` mit der Dauer, die in `PT` definiert ist, sobald der Eingang `IN` aktiviert wird.

4. Hardware-Ausgang

- **Name**: `DigitalOutput_Q1`
- **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
- **Parameter**:
  - `Output` = `Output_Q1` (Verweis auf den physikalischen Ausgang Q1)
- **Funktionsweise**: Schaltet den physikalischen Ausgang basierend auf dem Logiksignal.

Programmablauf und Verbindungen

Der Ablauf der Schaltung gestaltet sich wie folgt:

1. **Signaleingang**: Das Signal von `DigitalInput_I1` (Eingang I1) wird an den `CLK`-Eingang des Flankenerkennungsbausteins `AX_FB_R_TRIG` geleitet.
2. **Flankenauswertung**:
   - Wenn der Taster an I1 gedrückt wird, erkennt `AX_FB_R_TRIG` die positive Flanke.
   - Der Ausgang `Q` des Triggers wird kurzzeitig aktiv.
3. **Zeitsteuerung**: Dieses Signal wird an den Eingang `IN` des Timers `AX_TP` weitergeleitet.
4. **Ausgabe**: Der Timer aktiviert seinen Ausgang `Q` für genau **1 Sekunde** (`PT=T#1s`). Dieses Signal steuert den `DigitalOutput_Q1` an.

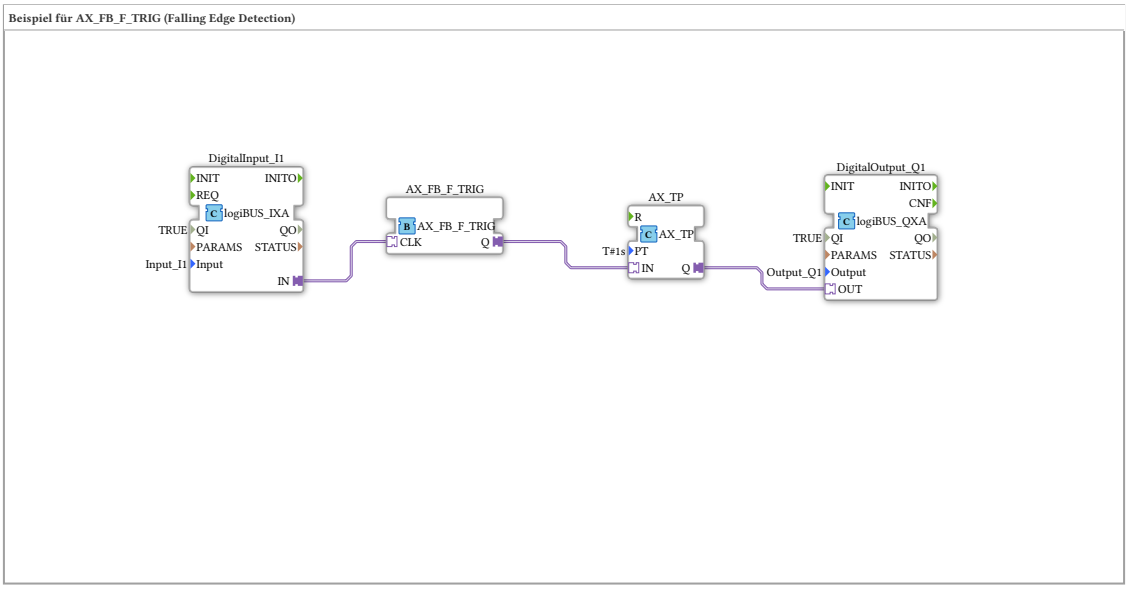
Zusammenhang der Verbindungen:

- `DigitalInput_I1.IN` ⟶ `AX_FB_R_TRIG.CLK`
- `AX_FB_R_TRIG.Q` ⟶ `AX_TP.IN`
- `AX_TP.Q` ⟶ `DigitalOutput_Q1.OUT`

Zusammenfassung

Die `Uebung_177_AX` zeigt eine klassische Anwendung in der Automatisierungstechnik: Das Entkoppeln eines mechanischen Tastendrucks von der Ausführungsdauer einer Aktion. Durch die Verwendung von `AX_FB_R_TRIG` spielt es keine Rolle, wie lange der Taster gedrückt gehalten wird; der Prozess wird nur einmal beim Drücken gestartet. Der Timer `AX_TP` sorgt dafür, dass der Ausgang (z.B. eine Lampe oder ein Motor) für eine exakte Zeitspanne (hier 1 Sekunde) läuft.

2.1.2.467 Uebung_178_AX: Beispiel für AX_FB_F_TRIG (Falling Edge Detection)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert die Verwendung des Funktionsbausteins `AX_FB_F_TRIG` zur Erkennung einer fallenden Flanke (Falling Edge Detection). Ziel der Schaltung ist es, ein Ereignis oder Signal genau in dem Moment auszulösen, wenn ein Eingangssignal von einem hohen Status (TRUE) auf einen niedrigen Status (FALSE) wechselt (z.B. beim Loslassen eines Tasters).

Verwendete Funktionsbausteine (FBs)

In dieser Sub-Application werden verschiedene Funktionsbausteine kombiniert, um die Logik abzubilden. Da es sich hierbei um eine SubApp handelt, werden im Folgenden die intern verschalteten Bausteine beschrieben.

Sub-Bausteine: Interne Logik

Diese Übung besteht aus einer Vernetzung der folgenden Standard- und Adapter-Bausteine:

- **Verwendete interne FBs**:
  - **DigitalInput_I1**: `logiBUS::io::DI::logiBUS_IXA`
    - **Funktion**: Liest den physikalischen Eingang `Input_I1` ein.
    - **Parameter**:
      - `Input` = `Input_I1`
      - `QI` = `TRUE`
  - **AX_FB_F_TRIG**: `adapter::iec61131::edgeDetection::AX_FB_F_TRIG`
    - **Funktion**: Erkennt eine fallende Flanke am Eingangssignal. Der Ausgang wird aktiv, wenn der Eingang von TRUE auf FALSE wechselt.
  - **AX_TP**: `adapter::events::unidirectional::timers::AX_TP`
    - **Funktion**: Timer Pulse (Impulsgeber). Erzeugt einen Impuls definierter Länge, sobald der Eingang aktiviert wird.
    - **Parameter**:
      - `PT` = `T#1s` (Impulsdauer von 1 Sekunde).
  - **DigitalOutput_Q1**: `logiBUS::io::DQ::logiBUS_QXA`
    - **Funktion**: Steuert den physikalischen Ausgang `Output_Q1` an.
    - **Parameter**:
      - `Output` = `Output_Q1`
      - `QI` = `TRUE`

Programmablauf und Verbindungen

Der Ablauf des Programms lässt sich wie folgt beschreiben:

1. **Signaleingang**: Der digitale Eingang `DigitalInput_I1` überwacht den Status des Hardware-Eingangs `Input_I1`.
2. **Flankenerkennung**: Das Signal wird über eine Adapter-Verbindung an den Baustein `AX_FB_F_TRIG` (Eingang `CLK`) weitergeleitet. Dieser Baustein überwacht das Signal auf eine negative Flanke. Das bedeutet, er reagiert genau dann, wenn das Signal von 1 (TRUE) auf 0 (FALSE) abfällt (z.B. wenn ein Taster losgelassen wird).
3. **Zeitglied**: Sobald die fallende Flanke erkannt wird, sendet der `AX_FB_F_TRIG` über seinen Ausgang `Q` ein Signal an den Timer `AX_TP` (Eingang `IN`).
4. **Ausgabe**: Der Timer `AX_TP` ist als Impulsgeber konfiguriert (`TP`). Durch das Eingangssignal wird der Timer gestartet und setzt seinen Ausgang `Q` für die Dauer von 1 Sekunde (`PT` = `T#1s`) auf TRUE.
5. **Hardware-Ansteuerung**: Das Ausgangssignal des Timers wird an den `DigitalOutput_Q1` weitergeleitet, welcher den Hardware-Ausgang `Output_Q1` für die Dauer des Impulses aktiviert.

Zusammenfassend: Wenn der Eingang I1 ausgeschaltet wird (fallende Flanke), wird der Ausgang Q1 für genau eine Sekunde eingeschaltet.

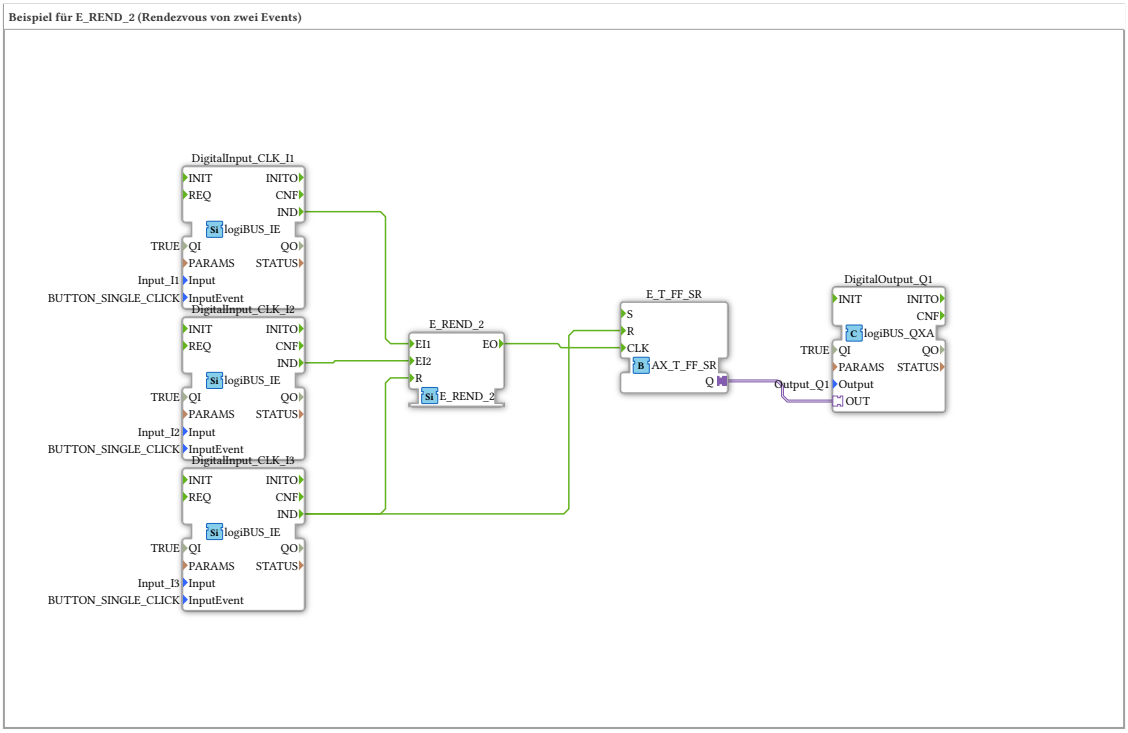
Zusammenfassung

Die Übung `Uebung_178_AX` veranschaulicht die Verarbeitung von Signalen basierend auf deren Abschaltmoment. Durch die Kombination einer fallenden Flankenerkennung (`F_TRIG`) mit einem Impuls-Timer

(TP) wird eine zeitgesteuerte Reaktion auf das Ende eines Eingangssignals realisiert. Dies ist eine typische Anwendung für Nachlaufsteuerungen oder Reaktionen auf das Loslassen von Bedienelementen.

2.1.2.468 Uebung_179_AX: Beispiel für E_REND_2 (Rendezvous von zwei Events)

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_179_AX (Beispiel für E_REND_2 (Rendezvous von zwei Events)).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Beispiel für E_REND_2 (Rendezvous von zwei Events)**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_179_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
- **DigitalInput_CLK_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **E_T_FF_SR**: Instanz des Typs adapter::events::unidirectional::AX_T_FF_SR.
- **E_REND_2**: Instanz des Typs iec61499::events::E_REND_2.
- **DigitalInput_CLK_I2**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I2
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **DigitalInput_CLK_I3**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I3
  - Parameter InputEvent = BUTTON_SINGLE_CLICK

Verbindungen und Schnittstellen

Adapterverbindungen:

- E_T_FF_SR.Q -> DigitalOutput_Q1.OUT

Ereignisverbindungen:

- DigitalInput_CLK_I1.IND -> E_REND_2.EI1
- DigitalInput_CLK_I2.IND -> E_REND_2.EI2
- DigitalInput_CLK_I3.IND -> E_REND_2.R
- E_REND_2.EO -> E_T_FF_SR.CLK
- DigitalInput_CLK_I3.IND -> E_T_FF_SR.R

Programmablauf und Funktionsweise

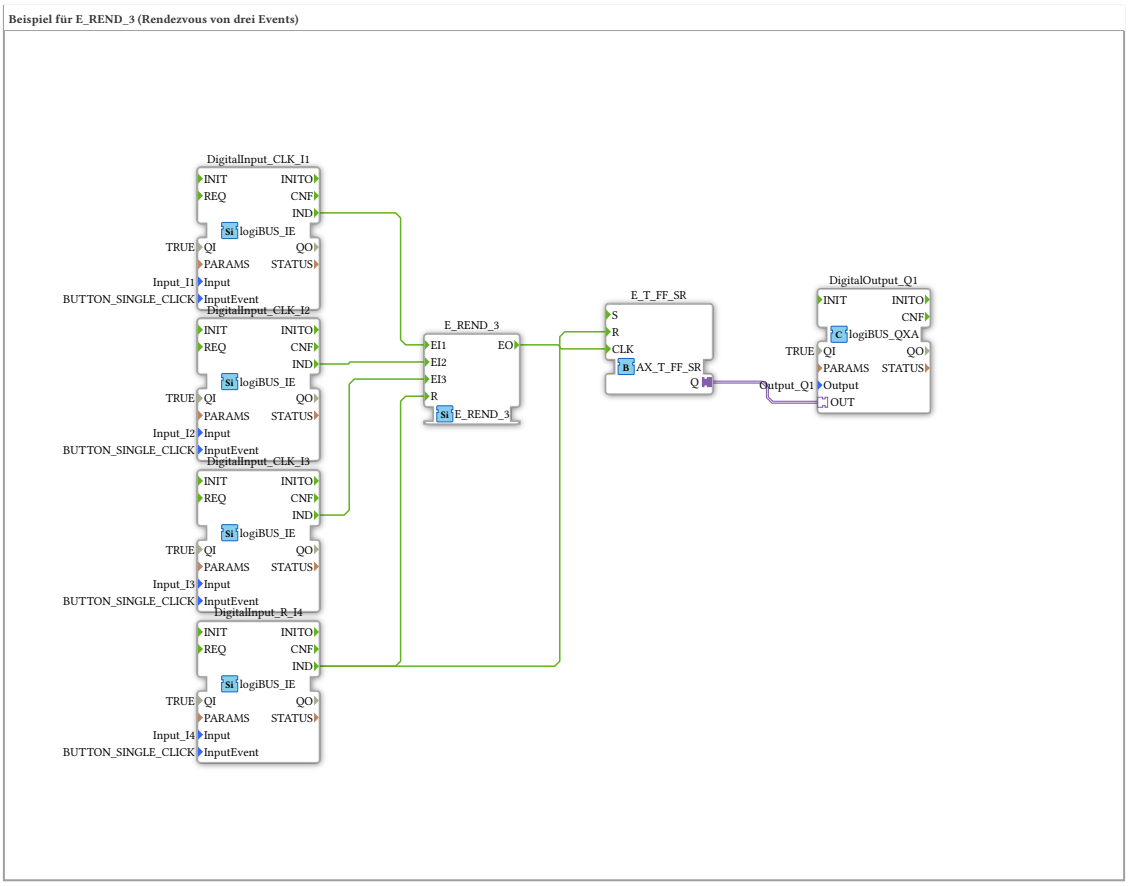
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_179_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.469 Uebung_180_AX: Beispiel für E_REND_3 (Rendezvous von drei Events)

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_180_AX (Beispiel für E_REND_3 (Rendezvous von drei Events)).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Beispiel für E_REND_3 (Rendezvous von drei Events)**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_180_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
- **DigitalInput_CLK_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **DigitalInput_CLK_I2**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I2
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **DigitalInput_CLK_I3**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I3
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **DigitalInput_R_I4**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I4
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **E_REND_3**: Instanz des Typs iec61499::events::E_REND_3.
- **E_T_FF_SR**: Instanz des Typs adapter::events::unidirectional::AX_T_FF_SR.

Verbindungen und Schnittstellen

Adapterverbindungen:

- E_T_FF_SR.Q -> DigitalOutput_Q1.OUT

Ereignisverbindungen:

- DigitalInput_CLK_I1.IND -> E_REND_3.EI1
- DigitalInput_CLK_I2.IND -> E_REND_3.EI2
- DigitalInput_CLK_I3.IND -> E_REND_3.EI3
- DigitalInput_R_I4.IND -> E_REND_3.R
- E_REND_3.EO -> E_T_FF_SR.CLK
- DigitalInput_R_I4.IND -> E_T_FF_SR.R

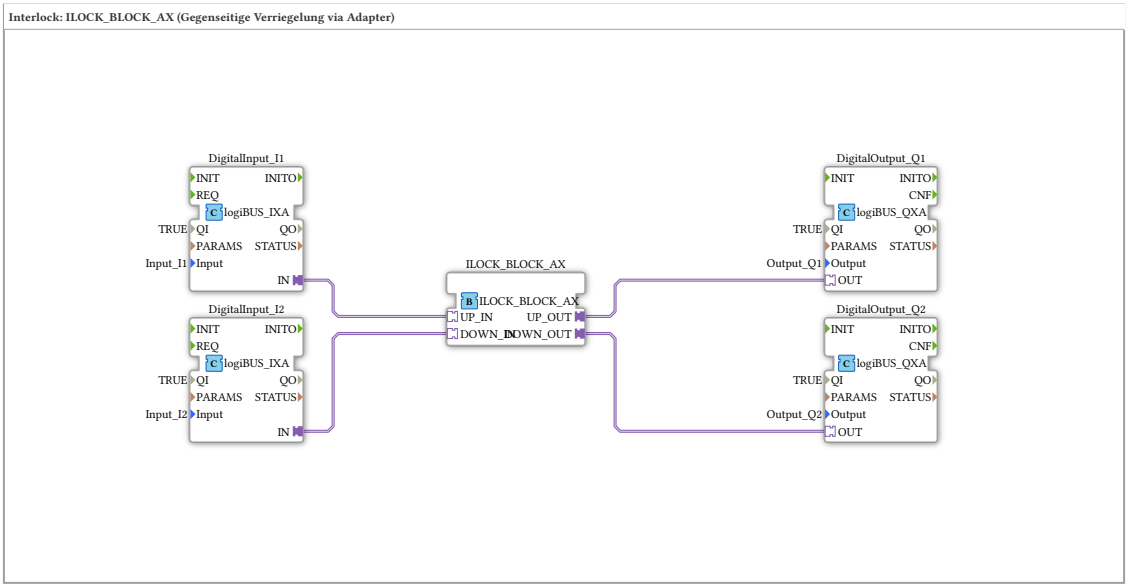
Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_180_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.470 Uebung_201_AX: Interlock: ILOCK_BLOCK_AX (Gegenseitige Verriegelung via Adapter)



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung demonstriert die Realisierung einer gegenseitigen Verriegelung (Interlock) mithilfe des Funktionsbausteins `ILOCK_BLOCK_AX`. Zwei digitale Eingänge werden über Adapter an den Verriegelungsblock angeschlossen. Die Ausgänge sind so geschaltet, dass immer nur einer der beiden Ausgänge aktiv sein kann – ein gleichzeitiges Schalten wird verhindert. Dies ist eine typische Sicherheitsfunktion in der Automatisierungstechnik, z. B. um gegenläufige Antriebe zu schützen.

Verwendete Funktionsbausteine (FBs)

Die Übung besteht aus fünf Funktionsbausteinen im Netzwerk:

- **DigitalInput_I1** – liest den ersten digitalen Eingang (`Input_I1`)
- **DigitalInput_I2** – liest den zweiten digitalen Eingang (`Input_I2`)
- **ILOCK_BLOCK_AX** – führt die gegenseitige Verriegelung durch
- **DigitalOutput_Q1** – steuert den ersten digitalen Ausgang (`Output_Q1`)
- **DigitalOutput_Q2** – steuert den zweiten digitalen Ausgang (`Output_Q2`)

Baustein: `DigitalInput_I1` (Typ: `logiBUS::io::DI::logiBUS_IXA`)

- **Typ**: Digitaleingabe-Baustein (Adapter-Schnittstelle)
- **Parameter**:
  - `QI = TRUE`
  - `Input = Input_I1`
- **Ereignisse/Adapter**: Gibt ein Adapter-Interface `IN` aus, das mit dem Interlock-Block verbunden wird.

Baustein: `DigitalInput_I2` (Typ: `logiBUS::io::DI::logiBUS_IXA`)

- **Typ**: Digitaleingabe-Baustein (Adapter-Schnittstelle)
- **Parameter**:
  - `QI = TRUE`
  - `Input = Input_I2`
- **Ereignisse/Adapter**: Gibt ein Adapter-Interface `IN` aus.

Baustein: `ILOCK_BLOCK_AX` (Typ: `logiBUS::signalprocessing::interlock::ILOCK_BLOCK_AX`)

- **Typ**: Verriegelungslogik-Baustein (Adapter-basiert)
- **Parameter**: Keine expliziten Parameter in der XML
- **Adapter-Schnittstellen**:
  - `UP_IN` – Eingang für den ersten Kanal (verbunden mit `DigitalInput_I1.IN`)

  - `DOWN_IN` – Eingang für den zweiten Kanal (verbunden mit `DigitalInput_I2.IN`)
  - `UP_OUT` – Ausgang für den ersten Kanal (verbunden mit `DigitalOutput_Q1.OUT`)
  - `DOWN_OUT` – Ausgang für den zweiten Kanal (verbunden mit `DigitalOutput_Q2.OUT`)
- **Funktionsweise**: Der Baustein implementiert eine gegenseitige Verriegelung. Wenn der `UP_IN`-Eingang aktiv ist, wird `UP_OUT` aktiviert und gleichzeitig `DOWN_OUT` deaktiviert (sperrt den zweiten Kanal). Wird `DOWN_IN` aktiv, schaltet der Baustein entsprechend um. Ein gleichzeitiges Aktivieren beider Ausgänge ist nicht möglich.

Baustein: `DigitalOutput_Q1` (Typ: `logiBUS::io::DQ::logiBUS_QXA`)

- **Typ**: Digitalausgabe-Baustein (Adapter-Schnittstelle)
- **Parameter**:
  - `QI` = `TRUE`
  - `Output` = `Output_Q1`
- **Ereignisse/Adapter**: Empfängt ein Adapter-Interface `OUT` vom Interlock-Block.

Baustein: `DigitalOutput_Q2` (Typ: `logiBUS::io::DQ::logiBUS_QXA`)

- **Typ**: Digitalausgabe-Baustein (Adapter-Schnittstelle)
- **Parameter**:
  - `QI` = `TRUE`
  - `Output` = `Output_Q2`
- **Ereignisse/Adapter**: Empfängt ein Adapter-Interface `OUT` vom Interlock-Block.

Programmablauf und Verbindungen

Der Ablauf der Übung ist wie folgt:

1. Die beiden digitalen Eingangssignale `Input_I1` und `Input_I2` werden durch die `DigitalInput_I1`- bzw. `DigitalInput_I2`-Bausteine erfasst.
2. Die Adapterausgänge dieser Eingangsbausteine (`IN`) werden mit den entsprechenden Eingängen des `ILOCK_BLOCK_AX` verbunden:
   - `DigitalInput_I1.IN` ⟶ `ILOCK_BLOCK_AX.UP_IN`
   - `DigitalInput_I2.IN` ⟶ `ILOCK_BLOCK_AX.DOWN_IN`
3. Im `ILOCK_BLOCK_AX` wird die Verriegelungslogik ausgeführt:
   - Bei Aktivierung von `UP_IN` wird `UP_OUT` auf TRUE gesetzt und `DOWN_OUT` auf FALSE.
   - Bei Aktivierung von `DOWN_IN` wird `DOWN_OUT` auf TRUE gesetzt und `UP_OUT` auf FALSE.
   - Falls beide Eingänge gleichzeitig aktiv sind, wird durch die interne Logik ein definierter Vorrang (meist der zuerst erkannte) sichergestellt.
4. Die Ausgangsadapter des Interlock-Blocks werden mit den Ausgangsbausteinen verbunden:
   - `ILOCK_BLOCK_AX.UP_OUT` ⟶ `DigitalOutput_Q1.OUT`
   - `ILOCK_BLOCK_AX.DOWN_OUT` ⟶ `DigitalOutput_Q2.OUT`
5. Die Ausgangsbausteine geben die Signale an die physikalischen Ausgänge `Output_Q1` und `Output_Q2` weiter.

Lernziele:

- Kennenlernen des Interlock-Konzepts (gegenseitige Verriegelung)
- Arbeiten mit Adapter-basierten Funktionsbausteinen in 4diac
- Verständnis für Sicherheitslogik in der Automatisierung

Schwierigkeitsgrad: Einsteiger / Fortgeschrittene – Grundkenntnisse in IEC 61499 und Adapterverbindungen sind hilfreich.

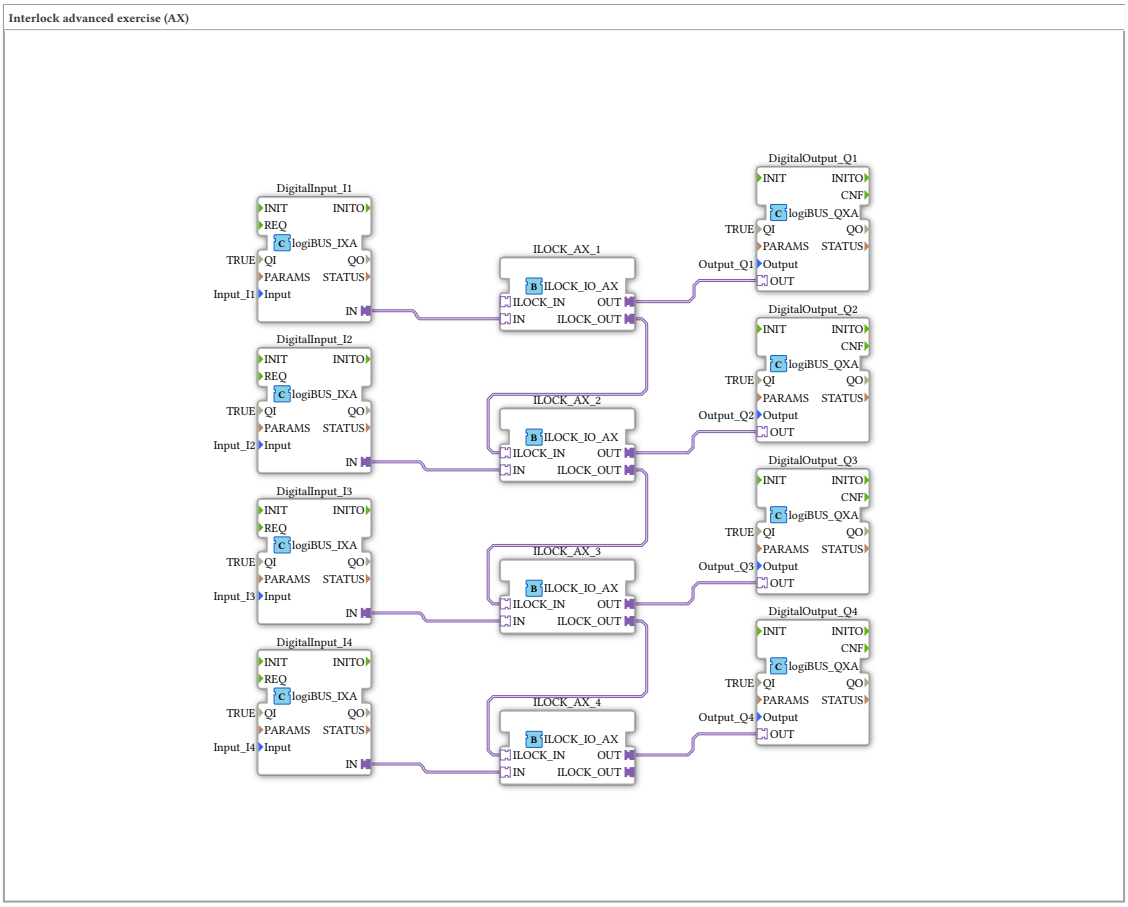
Zusammenfassung

In dieser Übung wird die gegenseitige Verriegelung zweier Ausgänge mithilfe des Funktionsbausteins `ILOCK_BLOCK_AX` umgesetzt. Die Struktur zeigt eine typische Anwendung von Adapterverbindungen zur Steuerung von Ein- und Ausgängen in der 4diac-IDE. Die Verriegelung verhindert, dass beide Ausgänge gleichzeitig aktiv werden – eine wichtige Sicherheitsfunktion für viele industrielle Steuerungsaufgaben.

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

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

2.1.2.471 Uebung_201_Interlock_AX: Interlock advanced exercise (AX)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung erweitert die grundlegende Interlock-Schaltung auf eine komplexere Anwendung (Advanced eXercise). Ziel ist es, vier digitale Eingänge (I1–I4) über eine verkettete Verriegelungslogik (Interlock) mit vier digitalen Ausgängen (Q1–Q4) zu verbinden. Die Besonderheit liegt in der seriellen Verschaltung der Interlock-Bausteine: Der Freigabeausgang eines Bausteins wird mit dem Freigabeeingang des nächsten Bausteins verbunden, sodass eine Abhängigkeitskette entsteht. Dies ermöglicht zeitliche oder logische Sperren zwischen aufeinanderfolgenden Ausgängen und eignet sich für Anwendungen wie sequenzielle Maschinensteuerungen oder Sicherheitsschaltungen.

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: DigitalInput_Ix

- **Typ**: `logiBUS::io::DI::logiBUS_IXA`
- **Verwendete interne FBs**: Keine (Hardware-nah)
- **Parameter**:
  - `QI` = `TRUE`
  - `Input` = entsprechendes physikalisches Eingangssignal (Input_I1–I4)
- **Funktionsweise**: Wandelt ein digitales Eingangssignal der logiBUS-Hardware in ein internes Adapter- oder Datensignal um. Stellt den Sensorwert am Adapterausgang `IN` bereit.

Sub-Bausteine: DigitalOutput_Qx

- **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
- **Verwendete interne FBs**: Keine (Hardware-nah)
- **Parameter**:
  - `QI` = `TRUE`
  - `Output` = entsprechendes physikalisches Ausgangssignal (Output_Q1–Q4)

- **Funktionsweise**: Empfängt ein Signal am Adaptereingang `OUT` und aktiviert den zugehörigen digitalen Hardware-Ausgang (z. B. Relais, Ventil, Lampe).

Sub-Baustein: ILOCK_AX_x

- **Typ**: `logiBUS::signalprocessing::interlock::ILOCK_IO_AX`
- **Verwendete interne FBs**: Keine (geschlossener Funktionsbaustein)
- **Parameter**: Keine expliziten Parameter in diesem Projekt (Standardparameter)
- **Adaptereingänge**:
  - `IN`: Digitaleingangsfreigabe-Schnittstelle
  - `ILOCK_IN`: Interlock-Verkettungseingang (von vorherigem Baustein)
- **Adapterausgänge**:
  - `OUT`: Digitalausgangsfreigabe-Schnittstelle
  - `ILOCK_OUT`: Interlock-Verkettungsausgang (zum nächsten Baustein)
- **Funktionsweise**: Der Baustein realisiert eine verriegelnde Steuerlogik (Interlock). Der Ausgang `OUT` wird nur dann aktiviert, wenn der Eingang `IN` aktiv *und* der Verkettungseingang `ILOCK_IN` freigegeben ist. Nach Aktivierung gibt er selbst am `ILOCK_OUT` die Freigabe für den nächsten Baustein in der Kette weiter. So entsteht eine sequenzielle Abhängigkeit: Q1 muss zuerst kommen, damit Q2 freigegeben wird, dann Q3, dann Q4.

Programmablauf und Verbindungen

Die Schaltung besteht aus vier identischen Interlock-Stufen, die in Reihe geschaltet sind. Jede Stufe enthält:

- einen digitalen Eingang (DigitalInput)
- einen ILOCK_IO_AX-Baustein
- einen digitalen Ausgang (DigitalOutput)

Die Verbindungen im Netzwerk:

1. **Eingangsseite**: Jeder `DigitalInput` ist mit seinem zugehörigen `ILOCK_AX`-Baustein über den Adapter `IN` verbunden.
   - `DigitalInput_I1.IN` ⟶ `ILOCK_AX_1.IN`
   - `DigitalInput_I2.IN` ⟶ `ILOCK_AX_2.IN`
   - `DigitalInput_I3.IN` ⟶ `ILOCK_AX_3.IN`
   - `DigitalInput_I4.IN` ⟶ `ILOCK_AX_4.IN`
2. **Ausgangsseite**: Jeder `ILOCK_AX` leitet sein Freigabesignal über den Adapter `OUT` an den zugehörigen `DigitalOutput` weiter.
   - `ILOCK_AX_1.OUT` ⟶ `DigitalOutput_Q1.OUT`
   - `ILOCK_AX_2.OUT` ⟶ `DigitalOutput_Q2.OUT`
   - `ILOCK_AX_3.OUT` ⟶ `DigitalOutput_Q3.OUT`
   - `ILOCK_AX_4.OUT` ⟶ `DigitalOutput_Q4.OUT`
3. **Verkettete Verriegelung (Interlock-Kette)**:
   - `ILOCK_AX_1.ILOCK_OUT` ⟶ `ILOCK_AX_2.ILOCK_IN`
   - `ILOCK_AX_2.ILOCK_OUT` ⟶ `ILOCK_AX_3.ILOCK_IN`
   - `ILOCK_AX_3.ILOCK_OUT` ⟶ `ILOCK_AX_4.ILOCK_IN`

Der Ablauf: Um Q2 zu aktivieren, muss zuerst Q1 aktiv sein (weil `ILOCK_IN` von Baustein 2 vom `ILOCK_OUT` von Baustein 1 kommt). Analog muss für Q3 sowohl Q1 als auch Q2 aktiv sein, für Q4 alle vorherigen. Diese Kette stellt sicher, dass die Ausgänge nur in der vorgegebenen Reihenfolge zugeschaltet werden können – ein typisches Sicherheits- oder Sequenzverhalten.

Hinweise zur Bedienung: Die Übung kann in der 4diac-IDE durch das Laden des SubApp-Typs `Uebung_201_Interlock_AX` und anschließendes Ausführen auf einer logiBUS-Hardwareplattform getestet werden. Die Eingänge sollten wechselweise oder nacheinander aktiviert werden, um die sequenzielle Freigabe zu beobachten.

Zusammenfassung

Die Übung `Uebung_201_Interlock_AX` demonstriert die praktische Anwendung verketteter Interlock-Bausteine (`ILOCK_IO_AX`) in einer 4diac-Umgebung. Vier unabhängige Eingangs-/Ausgangspaare werden seriell so verschaltet, dass jeder nachfolgende Ausgang nur freigegeben wird, wenn der vorherige Aus-

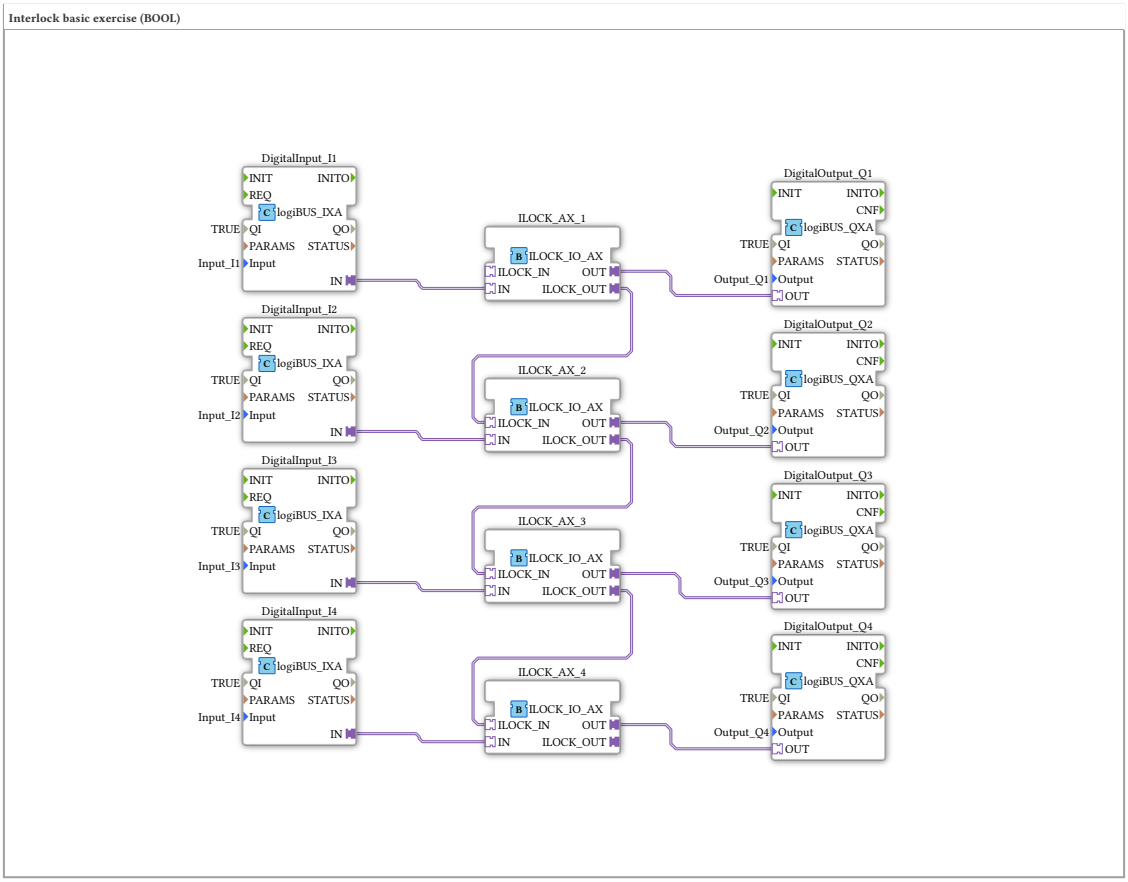
gang bereits aktiv ist. Dieses Prinzip wird häufig in der Automatisierungstechnik für Anlaufsequenzen, Schutzverriegelungen oder Maschinensicherheit eingesetzt. Der Lernende vertieft das Verständnis für die Verschaltung von Adaptern und die Erstellung von Abhängigkeitsketten in IEC 61499.

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

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

2.1.2.472 Uebung_201_Interlock_BOOL_AX: Interlock basic exercise (BOOL)

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_201_Interlock_BOOL_AX (Interlock basic exercise (BOOL)).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Interlock basic exercise (BOOL)**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_201_Interlock_BOOL_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
- **DigitalOutput_Q4**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q4
- **DigitalInput_I2**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I2

- **DigitalOutput_Q3**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q3
- **DigitalInput_I3**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I3
- **DigitalOutput_Q2**: Instanz des Typs logiBUS::io::DQ::logiBUS_QXA.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q2
- **ILOCK_AX_3**: Instanz des Typs logiBUS::signalprocessing::interlock::ILOCK_IO_AX.
- **ILOCK_AX_4**: Instanz des Typs logiBUS::signalprocessing::interlock::ILOCK_IO_AX.
- **ILOCK_AX_1**: Instanz des Typs logiBUS::signalprocessing::interlock::ILOCK_IO_AX.
- **ILOCK_AX_2**: Instanz des Typs logiBUS::signalprocessing::interlock::ILOCK_IO_AX.
- **DigitalInput_I4**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I4
- **DigitalInput_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1

Verbindungen und Schnittstellen

Adapterverbindungen:

- DigitalInput_I3.IN -> ILOCK_AX_3.IN
- DigitalInput_I4.IN -> ILOCK_AX_4.IN
- ILOCK_AX_4.OUT -> DigitalOutput_Q4.OUT
- ILOCK_AX_3.OUT -> DigitalOutput_Q3.OUT
- DigitalInput_I1.IN -> ILOCK_AX_1.IN
- ILOCK_AX_1.OUT -> DigitalOutput_Q1.OUT
- DigitalInput_I2.IN -> ILOCK_AX_2.IN
- ILOCK_AX_2.OUT -> DigitalOutput_Q2.OUT
- ILOCK_AX_1.ILOCK_OUT -> ILOCK_AX_2.ILOCK_IN
- ILOCK_AX_2.ILOCK_OUT -> ILOCK_AX_3.ILOCK_IN
- ILOCK_AX_3.ILOCK_OUT -> ILOCK_AX_4.ILOCK_IN

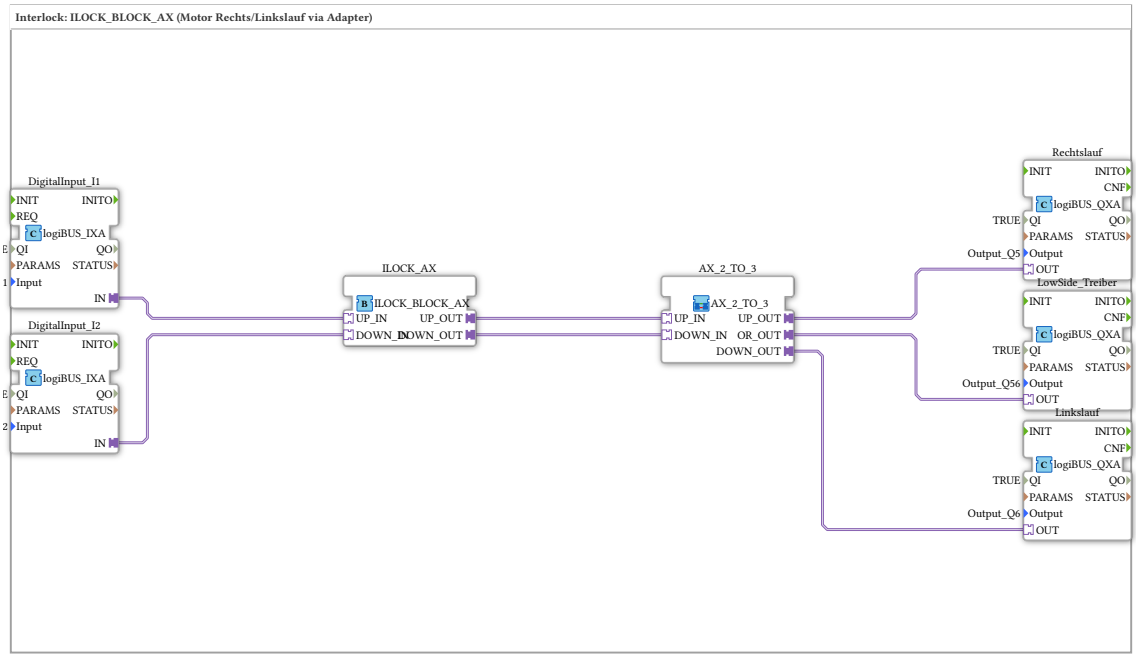
Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_201_Interlock_BOOL_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.473 Uebung_201b_AX: Interlock: ILOCK_BLOCK_AX (Motor Rechts/Linkslauf via Adapter)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert die Steuerung eines Motors mit Rechts- und Linkslauf unter Verwendung einer Interlock-Schaltung (Verriegelung). Der Funktionsbaustein `ILOCK_BLOCK_AX` verhindert, dass beide Drehrichtungen gleichzeitig aktiv werden. Die Eingangssignale stammen von zwei digitalen Sensoren (I1 und I2) über logiBUS-Digitalsignal-Adapter. Die Ausgänge steuern über logiBUS-Ausgangsbausteine den Motor (Rechtslauf Q5, Linkslauf Q6) sowie einen gemeinsamen Low-Side-Treiber (Q56). Die Signalanpassung wird durch den Sub-Applikationsbaustein `AX_2_TO_3` realisiert.

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_I1** / **DigitalInput_I2** Typ: `logiBUS::io::DI::logiBUS_IXA` Parametrisiert mit den physikalischen Eingängen `Input_I1` bzw. `Input_I2`. Diese Bausteine wandeln die binären Sensorsignale in Adapter-Signale um.
- **ILOCK_AX** Typ: `logiBUS::signalprocessing::interlock::ILOCK_BLOCK_AX` Zentraler Interlock-Baustein. Er erhält zwei Eingänge (`UP_IN`, `DOWN_IN`) und gibt zwei Ausgänge (`UP_OUT`, `DOWN_OUT`) weiter – jedoch nur dann, wenn nicht beide Eingänge gleichzeitig aktiv sind. Dadurch wird eine gegenseitige Verriegelung der Drehrichtungen sichergestellt.
- **Rechtslauf** / **Linkslauf** Typ: `logiBUS::io::DQ::logiBUS_QXA` Parametrisiert mit den Ausgängen `Output_Q5` (Rechtslauf) und `Output_Q6` (Linkslauf). Diese Bausteine schalten die entsprechenden Motorphasen.
- **LowSide_Treiber** Typ: `logiBUS::io::DQ::logiBUS_QXA` Parametrisiert mit `Output_Q56`. Dieser Ausgang aktiviert den gemeinsamen Low-Side-Schalter (z. B. Masseverbindung für den Motor).

Sub-Bausteine: `AX_2_TO_3`

- **Typ**: `MyLib::sys::AX_2_TO_3` (SubApplikation, keine eigenständige FB-Deklaration)
- **Verwendete interne FBs**: Die interne Struktur ist in dieser Übung nicht weiter aufgeschlüsselt. Es handelt sich um eine logische Umsetzung, die zwei Adapter-Eingänge (`UP_IN`, `DOWN_IN`) in drei Ausgangssignale (`UP_OUT`, `DOWN_OUT`, `OR_OUT`) aufteilt.
- **Funktionsweise**:
  - `UP_IN` ⟶ `UP_OUT` (Rechtslauf-Signal)
  - `DOWN_IN` ⟶ `DOWN_OUT` (Linkslauf-Signal)

- Die ODER-Verknüpfung beider Eingänge erzeugt das Signal für den Low-Side-Treiber (`OR_OUT`), da der Motor bei jeder Drehrichtung eine gemeinsame Masse benötigt. Die genaue Logik (z. B. Flankenverarbeitung oder Verzögerung) bleibt dem Hersteller des Sub-Bausteins vorbehalten.

Programmablauf und Verbindungen

1. **Digitale Eingänge**: Die Sensoren an `Input_I1` und `Input_I2` werden über `DigitalInput_I1` und `DigitalInput_I2` als Adapter-Signale bereitgestellt.
2. **Interlock**: Diese Signale gehen an die Adapter-Eingänge `UP_IN` und `DOWN_IN` des `ILOCK_BLOCK_AX`. Nur wenn nicht beide gleichzeitig aktiv sind, werden die Signale nach `UP_OUT` bzw. `DOWN_OUT` durchgeschaltet.
3. **Signalumsetzung**: Die Ausgänge des Interlock-Bausteins (`UP_OUT`, `DOWN_OUT`) werden mit den entsprechenden Eingängen des Sub-Applikationsbausteins `AX_2_TO_3` verbunden. Dieser wandelt die zwei Adapter-Signale in drei Ausgangssignale:
   - `UP_OUT` ⟶ Rechtslauf (an `Rechtslauf.OUT`)
   - `DOWN_OUT` ⟶ Linkslauf (an `Linkslauf.OUT`)
   - `OR_OUT` ⟶ Low-Side-Treiber (an `LowSide_Treiber.OUT`)
4. **Ausgangsbausteine**: Die drei logiBUS_QXA-Bausteine setzen die Adapter-Signale in physikalische Ausgänge an `Output_Q5`, `Output_Q56` und `Output_Q6` um.

Lernziele:

- Verständnis des Interlock-Prinzips für Motordrehrichtungen
- Umgang mit logiBUS-Ein-/Ausgangsadaptern
- Signalaufbereitung durch Sub-Applikationen
- Fehlervermeidung durch gegenseitige Verriegelung

Hinweise: Die Übung kann in der 4diac-IDE gestartet werden, nachdem die benötigten logiBUS-Bibliotheken importiert wurden. Der gesamte Ablauf ist Echtzeit-fähig und simuliert eine sichere Motorsteuerung.

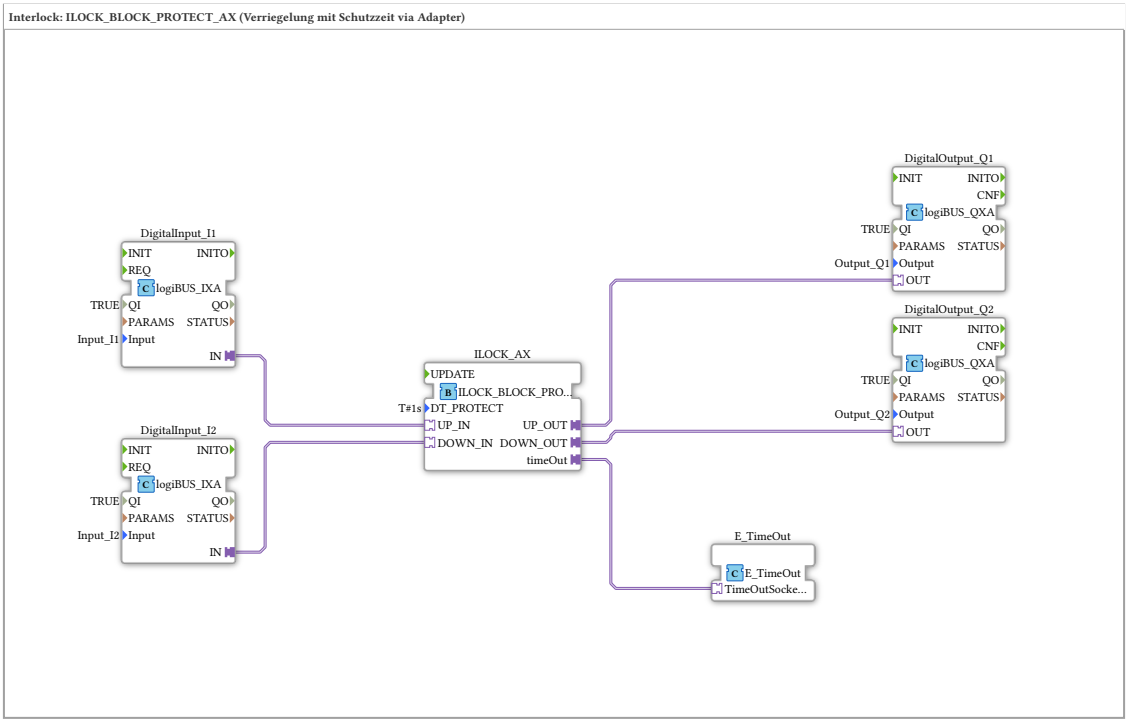
Zusammenfassung

Die Übung `Uebung_201b_AX` realisiert eine Interlock-gesteuerte Motorsteuerung mit Rechts- und Linkslauf. Kern ist der `ILOCK_BLOCK_AX`, der eine gleichzeitige Aktivierung beider Drehrichtungen verhindert. Die Adapter-basierten Ein- und Ausgänge werden über logiBUS-Bausteine an die Peripherie angebunden. Ein Sub-Applikationsbaustein (`AX_2_TO_3`) sorgt für die korrekte Verteilung der Signale auf drei Ausgänge (Rechtslauf, Linkslauf, Low-Side-Treiber). Die Schaltung ist ein einfaches, aber praxisnahes Beispiel für Verriegelungslogik in der Automatisierungstechnik.

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

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

2.1.2.474 Uebung_202_AX: Interlock: ILOCK_BLOCK_PROTECT_AX (Verriegelung mit Schutzzeit via Adapter)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert die Implementierung einer Verriegelung (Interlock) mit Schutzzeit unter Verwendung von Adaptern. Der Funktionsbaustein `ILOCK_BLOCK_PROTECT_AX` wird eingesetzt, um zwei Eingangssignale (z. B. Schalter oder Sensoren) gegenseitig zu verriegeln und über eine konfigurierbare Schutzzeit zu überwachen. Die Ausgänge steuern entsprechende Aktoren. Ein zusätzlicher Timer (`E_TimeOut`) zeigt an, wenn die Schutzzeit abläuft.

Das Netzwerk ist als SubAppType realisiert und kann in übergeordnete Applikationen eingebunden werden.

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine

DigitalInput_I1 / DigitalInput_I2 (je `logiBUS::io::DI::logiBUS_IXA`)

- **Typ**: logiBUS Digital Input Adapter
- **Parameter**:
  - `QI` = TRUE
  - `Input` = `Input_I1` (bzw. `Input_I2`)
- **Funktionsweise**: Stellt den physikalischen digitalen Eingang als Adapter-Socket zur Verfügung. Die eingehenden Signale werden über die logiBUS-Hardware gelesen und stehen über den Ausgang `IN` für weitere Verbindungen bereit.

ILOCK_AX (`logiBUS::signalprocessing::interlock::ILOCK_BLOCK_PROTECT_AX`)

- **Typ**: Interlock-Block mit Schutzzeit (Adapter)
- **Parameter**:
  - `DT_PROTECT` = `T#1s` (Schutzzeit 1 Sekunde)
- **Verwendete interne FBs**: Keine (Blackbox)
- **Funktionsweise**: Realisiert eine gegenseitige Verriegelung zweier Eingänge (`UP_IN`, `DOWN_IN`) und gibt die entsprechenden Ausgänge (`UP_OUT`, `DOWN_OUT`) frei. Die Schutzzeit verhindert ein ungewolltes schnelles Umschalten. Bei Aktivierung der Schutzzeit wird zudem ein Ereignis am Ausgang `timeOut` ausgelöst.

DigitalOutput_Q1 / DigitalOutput_Q2 (je `logiBUS::io::DQ::logiBUS_QXA`)

- **Typ**: logiBUS Digital Output Adapter
- **Parameter**:
  - `QI` = TRUE
  - `Output` = `Output_Q1` (bzw. `Output_Q2`)
- **Funktionsweise**: Nimmt das Signal am Eingang `OUT` entgegen und gibt es über den logiBUS-Ausgangskanal an die angeschlossene Hardware weiter.

E_TimeOut (`iec61499::events::E_TimeOut`)

- **Typ**: Ereignis-Timer
- **Parameter**: Keine
- **Funktionsweise**: Ein einfacher Timer, der durch ein eingehendes Ereignis am Eingang `TimeOutSocket` gestartet wird und nach Ablauf der eingestellten Zeit ein Ausgangsereignis auslöst. Wird hier verwendet, um das Timeout-Signal des ILOCK-Blocks aufzufangen und einer weiteren Verarbeitung zuzuführen.

Programmablauf und Verbindungen

Die SubApp ist wie folgt verschaltet:

1. **Eingänge**: Die beiden logiBUS-Digital-Input-Adapter (`DigitalInput_I1`, `DigitalInput_I2`) lesen die Hardware-Signale von den Kanälen `Input_I1` und `Input_I2` ein. Ihre `IN`-Ausgänge sind über **Adapterverbindungen** mit den entsprechenden Eingängen des Interlock-Blocks verbunden:
   - `DigitalInput_I1.IN` ⟶ `ILOCK_AX.UP_IN`
   - `DigitalInput_I2.IN` ⟶ `ILOCK_AX.DOWN_IN`
2. **Interlock-Verarbeitung**: Der FB `ILOCK_AX` wertet die Signale aus. Solange keine Verriegelung aktiv ist, werden die Ausgänge `UP_OUT` bzw. `DOWN_OUT` entsprechend der Eingänge gesetzt. Überschreitet die Umschaltzeit die eingestellte Schutzzeit (`DT_PROTECT`), wird der Ausgang `timeOut` aktiv.
3. **Ausgänge**: Die freigegebenen Signale werden über Adapterverbindungen zu den logiBUS-Digital-Output-Adaptern geführt:
   - `ILOCK_AX.UP_OUT` ⟶ `DigitalOutput_Q1.OUT`
   - `ILOCK_AX.DOWN_OUT` ⟶ `DigitalOutput_Q2.OUT` Die Ausgänge `Output_Q1` und `Output_Q2` steuern die angeschlossenen Aktoren.
4. **Timer**: Das Timeout-Ereignis des Interlock-Blocks (`ILOCK_AX.timeOut`) ist mit dem Socket `E_TimeOut.TimeOutSocket` verbunden. Der Timer kann in einer übergeordneten Applikation genutzt werden, um eine Fehler- oder Quittierungsmeldung zu generieren.

Lernziele der Übung:

- Verständnis des Interlock-Konzepts mit Schutzzeit
- Einsatz von logiBUS-IO-Adaptern in 4diac
- Verwendung von Ereignis-Timern (`E_TimeOut`)
- Fehlersuche und Analyse von Zeitproblemen in Steuerungen

Schwierigkeitsgrad: Mittel **Vorkenntnisse**: Grundlegende Bedienung der 4diac-IDE, Kenntnisse der logiBUS-Hardware, Umgang mit Funktionsbausteinen und Adaptern.

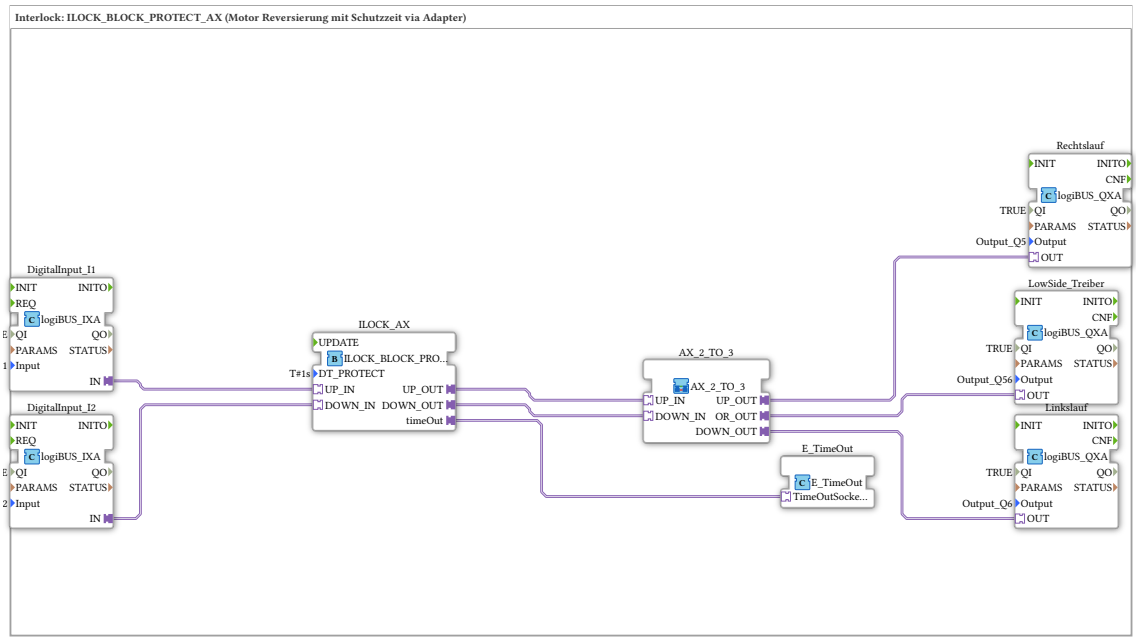
Zusammenfassung

Die Übung `Uebung_202_AX` veranschaulicht eine industrietypische Verriegelungsschaltung mit Schutzzeit auf Basis des Funktionsbausteins `ILOCK_BLOCK_PROTECT_AX`. Durch den Einsatz von logiBUS-Adaptern werden digitale Ein- und Ausgänge angebunden, während ein zusätzlicher Timer das Timeout-Ereignis verarbeitet. Das Zusammenspiel der Bausteine wird über Adapterverbindungen hergestellt, was eine flexible und modulare Struktur ermöglicht.

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

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

2.1.2.475 Uebung_202b_AX: Interlock: ILOCK_BLOCK_PROTECT_AX (Motor Reversierung mit Schutzzeit via Adapter)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung zeigt die Verwendung des Funktionsbausteins `ILOCK_BLOCK_PROTECT_AX` zur sicheren Ansteuerung eines Motors mit Reversierfunktion. Dabei wird eine **Umschaltverzögerung (Schutzzeit)** realisiert, die verhindert, dass der Motor sofort von einer Drehrichtung in die andere umschaltet. Zusätzlich wird ein **Low-Side-Treiber** für die gemeinsame Versorgung der Ausgänge eingesetzt. Die Logik ist als Subapplikation aufgebaut und nutzt einen Adapter-basierten Datenfluss.

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_I1** – Typ: `logiBUS::io::DI::logiBUS_IXA`
  - Parameter: QI = TRUE, Input = `Input_I1`
  - Wandelt ein digitales Eingangssignal (z. B. Taster für Aufwärts) in ein Logiksignal um.
- **DigitalInput_I2** – Typ: `logiBUS::io::DI::logiBUS_IXA`
  - Parameter: QI = TRUE, Input = `Input_I2`
  - Wandelt ein digitales Eingangssignal (z. B. Taster für Abwärts) in ein Logiksignal um.
- **ILOCK_AX** – Typ: `logiBUS::signalprocessing::interlock::ILOCK_BLOCK_PROTECT_AX`
  - Parameter: DT_PROTECT = `T#1s` (Schutzzeit von 1 Sekunde)
  - Kernbaustein der Übung: Erzeugt aus den beiden Eingangssignalen (`UP_IN`, `DOWN_IN`) verzögerte Ausgangssignale (`UP_OUT`, `DOWN_OUT`) und ein Zeitsignal (`timeOut`). Die Schutzzeit verhindert ein zu schnelles Umschalten.
- **LowSide_Treiber** – Typ: `logiBUS::io::DQ::logiBUS_QXA`
  - Parameter: QI = TRUE, Output = `Output_Q56`
  - Steuert einen gemeinsamen Low-Side-Ausgang (z. B. für die Freigabe der Motorbremse oder gemeinsame Versorgung).
- **Linkslauf** – Typ: `logiBUS::io::DQ::logiBUS_QXA`
  - Parameter: QI = TRUE, Output = `Output_Q6`
  - Schaltet den Motor für Linkslauf.
- **Rechtslauf** – Typ: `logiBUS::io::DQ::logiBUS_QXA`
  - Parameter: QI = TRUE, Output = `Output_Q5`

  - Schaltet den Motor für Rechtslauf.
- **E_TimeOut** – Typ: `iec61499::events::E_TimeOut`
  - Nimmt das Zeitsignal von `ILOCK_AX` entgegen (z. B. zur Visualisierung oder weiteren Verarbeitung).

Sub-Baustein: `AX_2_TO_3`
- **Typ**: `MyLib::sys::AX_2_TO_3` (Subapplikation)
- **Verwendete interne FBs**: nicht in dieser Übungsdatei definiert (gekapselte Logik)
- **Funktionsweise** (abgeleitet aus den Adapterverbindungen):
  - Empfängt die Signale `UP_IN` und `DOWN_IN` von `ILOCK_AX`.
  - Leitet `UP_IN` an `UP_OUT` und `DOWN_IN` an `DOWN_OUT` weiter (oder führt eine logische Verknüpfung durch).
  - Erzeugt ein ODER-Signal (`OR_OUT`) aus beiden Eingängen, das den **LowSide_Treiber** ansteuert.
  - Dient als Aufteilung der ILOCK-Ausgänge auf die einzelnen Motorausgänge und den gemeinsamen Freigabeausgang.

Programmablauf und Verbindungen

1. **Eingänge**
   - Die digitalen Eingänge `Input_I1` (Auf) und `Input_I2` (Ab) werden über die Bausteine `DigitalInput_I1` und `DigitalInput_I2` eingelesen.
   - Ihre Signale werden direkt an die Adaptereingänge `UP_IN` und `DOWN_IN` des `ILOCK_AX` weitergegeben.
2. **Interlock-Logik**
   - `ILOCK_AX` wertet die anstehenden Signale aus. Bei einem Wechsel von einer Drehrichtung in die andere wird die parametrierte **Schutzzeit DT_PROTECT = 1s** aktiv.
   - Erst nach Ablauf dieser Zeit wird das neue Ausgangssignal auf `UP_OUT` oder `DOWN_OUT` geschaltet.
   - Gleichzeitig wird das Zeitgebersignal `timeOut` für die Dauer der Schutzzeit auf `TRUE` gesetzt und an `E_TimeOut` übertragen.
3. **Signalverteilung über SubApp `AX_2_TO_3`**
   - Die verzögerten Ausgänge `UP_OUT` und `DOWN_OUT` von `ILOCK_AX` gelangen über Adapterverbindungen in die SubApp `AX_2_TO_3`.
   - Diese SubApp leitet die Signale an die entsprechenden Ausgänge `UP_OUT` und `DOWN_OUT` weiter und erzeugt ein ODER-Signal (`OR_OUT`), das den **LowSide_Treiber** aktiviert, sobald eine der beiden Drehrichtungen angefordert wird.
4. **Ausgänge**
   - `Rechtslauf` und `Linkslauf` werden direkt von den Ausgängen der SubApp angesteuert.
   - `LowSide_Treiber` wird über das ODER-Signal aktiviert und steuert den gemeinsamen Ausgang `Output_Q56`.

Lernziele
- Verständnis des Interlock-Konzepts für Motor-Reversierung
- Anwendung eines Schutzzeit-Bausteins (`ILOCK_BLOCK_PROTECT_AX`)
- Arbeiten mit Adapterverbindungen und Subapplikationen in 4diac
- Einbindung von Low-Side-Treibern in sicherheitsrelevante Steuerungen

Schwierigkeitsgrad: Fortgeschritten **Vorkenntnisse:** Grundlagen der 4diac-IDE, Umgang mit logiBUS-Bausteinen, Verständnis von Ereignis-/Datenflüssen

---

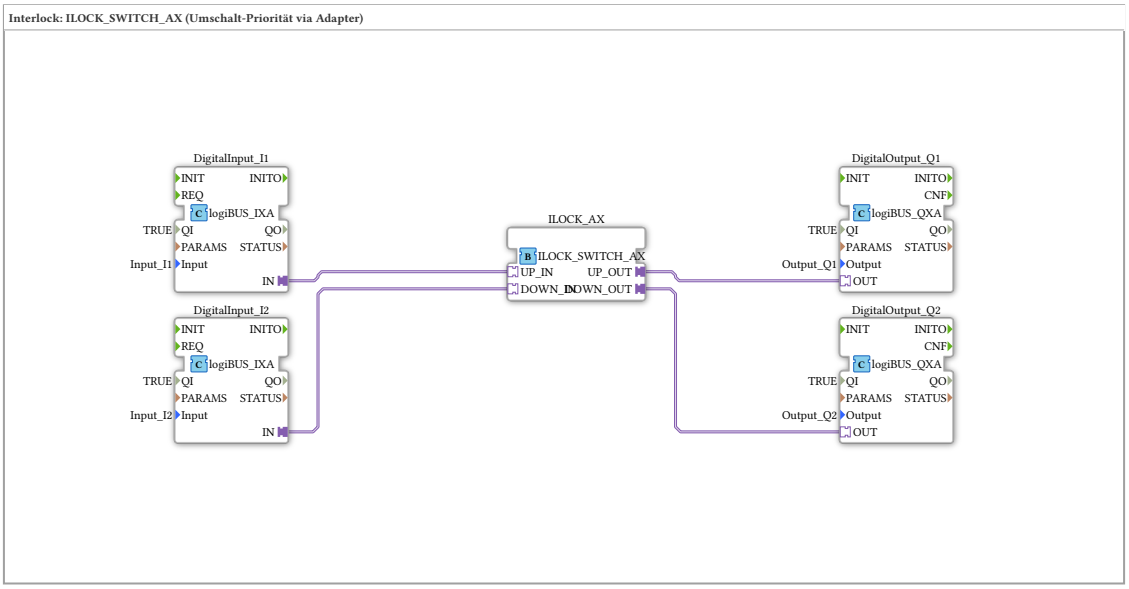
Zusammenfassung

In dieser Übung wird eine vollständige Motor-Reversiersteuerung mit **Umschaltverzögerung (Schutzzeit)** realisiert. Der Baustein `ILOCK_BLOCK_PROTECT_AX` übernimmt die sichere Verriegelung der Drehrichtungen, während die SubApp `AX_2_TO_3` die Signalverteilung auf die einzelnen Ausgänge und den gemeinsamen Low-Side-Treiber übernimmt. Die Übung vermittelt praxisnahe Kenntnisse zur sicheren Steuerung von Aktoren in der Automatisierungstechnik.

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

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

2.1.2.476 Uebung_203_AX: Interlock: ILOCK_SWITCH_AX (Umschalt-Priorität via Adapter)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert die Verwendung eines **Interlock-Funktionsbausteins mit Prioritätsumschaltung** (ILOCK_SWITCH_AX). Zwei digitale Eingänge (Input_I1, Input_I2) steuern über einen Adapter-basierten Interlock-Baustein zwei digitale Ausgänge (Output_Q1, Output_Q2). Der Interlock sorgt dafür, dass immer nur ein Ausgang aktiv sein kann – bei gleichzeitigen Eingangssignalen setzt sich ein definierter Prioritätsmechanismus durch. Die Kommunikation mit der Peripherie erfolgt über logiBUS-Adapterschnittstellen.

Verwendete Funktionsbausteine (FBs)

Die SubApp enthält fünf Funktionsbausteine aus der logiBUS-Bibliothek:

- **DigitalInput_I1** (Typ: `logiBUS::io::DI::logiBUS_IXA`)
  - Parameter: `QI = TRUE`, `Input = Input_I1`
  - Übernahme des digitalen Eingangssignals vom logiBUS-Kanal Input_I1.
- **DigitalInput_I2** (Typ: `logiBUS::io::DI::logiBUS_IXA`)
  - Parameter: `QI = TRUE`, `Input = Input_I2`
  - Übernahme des digitalen Eingangssignals vom logiBUS-Kanal Input_I2.
- **ILOCK_AX** (Typ: `logiBUS::signalprocessing::interlock::ILOCK_SWITCH_AX`)
  - Dieser Funktionsbaustein implementiert eine **Interlock-Funktion** mit zwei Eingangsadaptern (`UP_IN`, `DOWN_IN`) und zwei Ausgangsadaptern (`UP_OUT`, `DOWN_OUT`).
  - **Funktionsweise:**
    - Ist nur ein Eingang aktiv, wird der zugehörige Ausgang gesetzt.
    - Sind beide Eingänge gleichzeitig aktiv, entscheidet eine interne **Prioritätslogik** (in dieser Konfiguration: „UP“ hat Vorrang vor „DOWN“), welcher Ausgang gesetzt wird.
    - Der jeweils andere Ausgang wird zurückgesetzt.
  - Die Priorität kann über Parameter des Bausteins konfiguriert werden (hier Standard: UP priorisiert).
- **DigitalOutput_Q1** (Typ: `logiBUS::io::DQ::logiBUS_QXA`)
  - Parameter: `QI = TRUE`, `Output = Output_Q1`

  - Ausgabe des Signals auf den logiBUS-Ausgangskanal Output_Q1.
- **DigitalOutput_Q2** (Typ: `logiBUS::io::DQ::logiBUS_QXA`)
  - Parameter: `QI = TRUE`, `Output = Output_Q2`
  - Ausgabe des Signals auf den logiBUS-Ausgangskanal Output_Q2.

Programmablauf und Verbindungen

Die folgende Adapter-Verbindungsstruktur liegt dem Ablauf zugrunde:

1. **Eingangssignale:**
   - `DigitalInput_I1.IN` ⟶ `ILOCK_AX.UP_IN`
   - `DigitalInput_I2.IN` ⟶ `ILOCK_AX.DOWN_IN`
2. **Interlock-Verarbeitung:**
   - Der ILOCK_SWITCH_AX-Baustein wertet die eingehenden Signale aus und entscheidet nach Priorität, welcher Ausgang gesetzt wird.
3. **Ausgangssignale:**
   - `ILOCK_AX.UP_OUT` ⟶ `DigitalOutput_Q1.OUT`
   - `ILOCK_AX.DOWN_OUT` ⟶ `DigitalOutput_Q2.OUT`

Bei aktivem Signal an **Input_I1** wird **Output_Q1** gesetzt. Bei aktivem Signal an **Input_I2** wird **Output_Q2** gesetzt. Sind beide Eingänge gleichzeitig aktiv, erhält **Output_Q1** den Vorrang („UP priorisiert“). Dieses Verhalten ist typisch für sicherheitsrelevante Anwendungen, in denen eine gegenseitige Verriegelung von Aktoren (z. B. Richtungswechsel eines Motors) erforderlich ist.

Hinweise für die Durchführung:

- Die Übung richtet sich an Teilnehmer mit Grundkenntnissen der 4diac-IDE und des logiBUS-Adaptersystems.
- Lernziel ist das Verständnis von Interlock-Mechanismen über Adapter und die Prioritätssteuerung.
- Voraussetzung: Ein funktionierendes logiBUS-Projekt mit freien Ein-/Ausgangskanälen (Input_I1, Input_I2, Output_Q1, Output_Q2).
- Starten Sie die Übung, indem Sie die SubApp in Ihr System einbinden und die Klemmen entsprechend verdrahten.

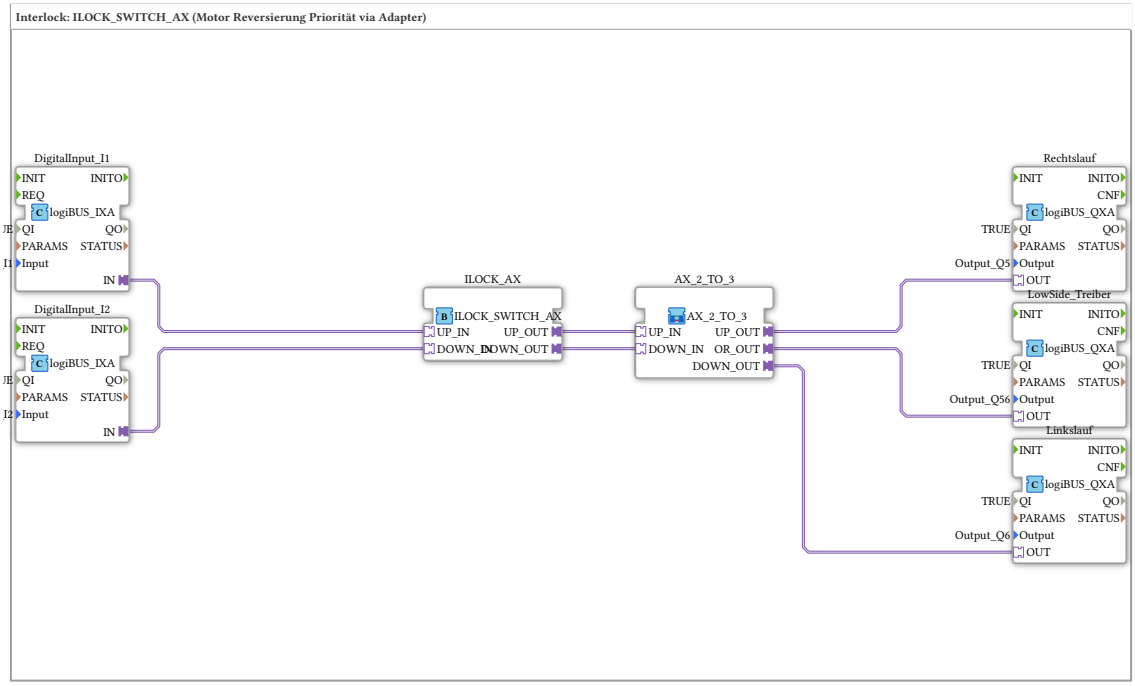
Zusammenfassung

Die Übung **Uebung_203_AX** vermittelt den Einsatz eines **prioritätsgesteuerten Interlocks** über Adapterschnittstellen. Zwei digitale Eingänge werden über den Baustein `ILOCK_SWITCH_AX` verarbeitet, der bei gleichzeitigen Signalen eine festgelegte Priorität (hier: UP vor DOWN) anwendet und die Ausgänge entsprechend schaltet. Die Implementierung erfolgt vollständig mit logiBUS-I/O-Bausteinen und zeigt eine typische Verriegelungsschaltung, wie sie in der Automatisierungstechnik häufig benötigt wird.

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

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

2.1.2.477 Uebung_203b_AX: Interlock: ILOCK_SWITCH_AX (Motor Reversierung Priorität via Adapter)



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung realisiert eine **Motorreversierung mit Interlock (Verriegelung)** unter Verwendung des Funktionsbausteins `ILOCK_SWITCH_AX`. Die Schaltung verhindert, dass beide Drehrichtungen gleichzeitig aktiviert werden, und gibt über einen Adapter die priorisierten Signale an eine nachgeschaltete Logik `AX_2_TO_3` weiter. Diese Logik wandelt die zwei Richtungssignale in drei Ausgänge um – einen für Rechtslauf, einen für Linkslauf und einen gemeinsamen Low-Side-Treiber, wie es für H-Brücken-Ansteuerungen typisch ist.

Zwei digitale Eingänge (I1, I2) dienen als Steuersignale für die Drehrichtung. Die Ausgänge Q5 (Rechtslauf), Q56 (Low-Side) und Q6 (Linkslauf) werden physisch über die logiBUS-Hardware angesteuert.

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: `AX_2_TO_3`

- **Typ**: `MyLib::sys::AX_2_TO_3`
- **Beschreibung**: Dieser Sub-Baustein (SubApp) dient als Verteilerlogik. Er nimmt zwei priorisierte Richtungssignale (`UP_IN`, `DOWN_IN`) von der vorhergehenden Interlock-Stufe entgegen und erzeugt daraus drei Ausgänge:
  - `UP_OUT` ⟶ Rechtslauf
  - `DOWN_OUT` ⟶ Linkslauf
  - `OR_OUT` ⟶ Low-Side-Treiber (aktiv, sobald eine Richtung aktiv ist)
- Die interne Implementierung ist in der Datei `AX_2_TO_3.subapp` definiert und wird in dieser Übung als Blackbox verwendet.

Weitere Funktionsbausteine

- **DigitalInput_I1**, **DigitalInput_I2**
  - **Typ**: `logiBUS::io::DI::logiBUS_IXA`
  - **Parameter**:
    - `QI` = `TRUE` (Freigabe)
    - `Input` = `Input_I1` bzw. `Input_I2`
  - **Datenausgang**: `OUT` (Adapter-Ausgang, der den digitalen Eingangszustand bereitstellt)
- **ILOCK_AX**

  - **Typ**: `logiBUS::signalprocessing::interlock::ILOCK_SWITCH_AX`
  - **Parameter**: keine
  - **Beschreibung**: Kern der Verriegelung. Er wertet die beiden Eingangssignale aus und gibt über `UP_OUT` und `DOWN_OUT` die jeweils aktive Richtung aus. Bei gleichzeitiger Aktivierung beider Eingänge setzt eine Prioritätslogik (definiert im Baustein) einen der Ausgänge zurück. Die Ausgänge sind Adapter-konform.

- **Rechtslauf**
  - **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
  - **Parameter**:
    - `QI` = `TRUE`
    - `Output` = `Output_Q5`
  - **Beschreibung**: Schaltet das Signal für den Rechtslauf auf den physischen Ausgang Q5.

- **LowSide_Treiber**
  - **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
  - **Parameter**:
    - `QI` = `TRUE`
    - `Output` = `Output_Q56`
  - **Beschreibung**: Schaltet den Low-Side-Treiber (gemeinsame Masse oder Bremse) auf den Ausgang Q56.

- **Linkslauf**
  - **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
  - **Parameter**:
    - `QI` = `TRUE`
    - `Output` = `Output_Q6`
  - **Beschreibung**: Schaltet das Signal für den Linkslauf auf den physischen Ausgang Q6.

Programmablauf und Verbindungen

Der logische Ablauf folgt einer klaren Kette:

1. **Eingangssignale**: Die beiden digitalen Eingänge (`Input_I1`, `Input_I2`) werden über die Bausteine `DigitalInput_I1` und `DigitalInput_I2` in Adapter-Signale umgewandelt.
2. **Interlock**: Diese Signale gelangen über die Adapter-Verbindungen an `ILOCK_AX`. Dort wird die Prioritätslogik angewendet. Die Ausgänge `UP_OUT` (für Rechtslauf) und `DOWN_OUT` (für Linkslauf) werden freigegeben – jedoch nie gleichzeitig.
3. **Signalverteilung**: Die priorisierten Signale werden an den Sub-Baustein `AX_2_TO_3` weitergeleitet. Dieser erzeugt aus den beiden Richtungssignalen drei Ausgänge:
   - `UP_OUT` ⟶ `Rechtslauf` (Q5)
   - `DOWN_OUT` ⟶ `Linkslauf` (Q6)
   - `OR_OUT` ⟶ `LowSide_Treiber` (Q56) – wird aktiv, sobald eine Richtung aktiv ist.
4. **Ausgangsstufen**: Die drei Ausgangsbausteine vom Typ `logiBUS_QXA` setzen die Signale auf die physikalischen Ausgänge der logiBUS-Hardware um.

Adapter-Verbindungen (in der XML als `AdapterConnections` definiert):

- `DigitalInput_I1.IN` ⟶ `ILOCK_AX.UP_IN`
- `DigitalInput_I2.IN` ⟶ `ILOCK_AX.DOWN_IN`
- `ILOCK_AX.UP_OUT` ⟶ `AX_2_TO_3.UP_IN`
- `ILOCK_AX.DOWN_OUT` ⟶ `AX_2_TO_3.DOWN_IN`
- `AX_2_TO_3.UP_OUT` ⟶ `Rechtslauf.OUT`
- `AX_2_TO_3.DOWN_OUT` ⟶ `Linkslauf.OUT`
- `AX_2_TO_3.OR_OUT` ⟶ `LowSide_Treiber.OUT`

Hinweise zur Durchführung:

- Diese Übung setzt Grundkenntnisse in der 4diac-IDE und dem Umgang mit Adaptern voraus.
- Schwierigkeitsgrad: **mittel**
- Lernziele: Verständnis von Interlock-Logiken, Arbeiten mit Sub-Applikationen und Adapter-basierter Signalweiterleitung, Realisierung einer sicheren Motoransteuerung.

- Um die Übung zu starten, muss die SubApp `Uebung_203b_AX` in ein Projekt eingebunden und mit den entsprechenden logiBUS-Eingängen und -Ausgängen verknüpft werden.

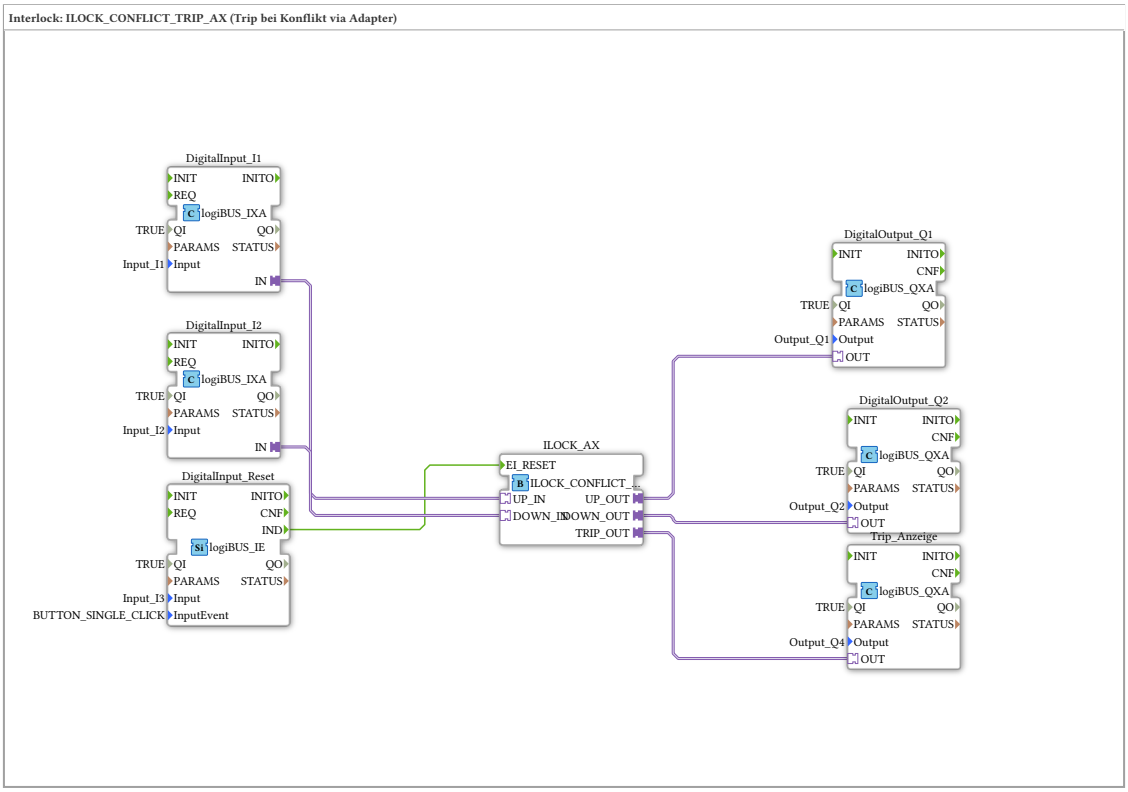
Zusammenfassung

Die Übung `Uebung_203b_AX` demonstriert eine vollständige Motorreversierung mit Verriegelung unter Verwendung des `ILOCK_SWITCH_AX`-Bausteins und einer nachgeschalteten Signalverteilung (`AX_2_TO_3`). Durch die Verwendung von Adaptern wird eine klare, modulare Struktur erreicht. Die Schaltung verhindert zuverlässig Kurzschlüsse oder Fehlansteuerungen und eignet sich für den Einsatz in der industriellen Steuerungstechnik.

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

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

2.1.2.478 Uebung_204_AX: Interlock: ILOCK_CONFLICT_TRIP_AX (Trip bei Konflikt via Adapter)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung implementiert eine Interlock-Logik mit Konflikterkennung und Trip-Auslösung unter Verwendung des Funktionsbausteins **ILOCK_CONFLICT_TRIP_AX**.
Zwei digitale Eingänge (über einen logiBUS-Adapter) melden Anforderungen in entgegengesetzte Richtungen (z. B. „Auf“ und „Ab“).
Tritt ein gleichzeitiges oder unzulässiges Signal auf, wird ein Trip ausgelöst, der über einen separaten Ausgang angezeigt wird. Ein Reset-Eingang (Einzelflanke) kann den Trip zurücksetzen.
Die Ausgänge steuern zwei digitale Aktoren (Q1, Q2) und eine Tripanzeige (Q4).

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: *keine*

Die gesamte Übung ist als eigenständige Subapplikation aufgebaut. Alle FBs werden direkt im Netzwerk eingesetzt.

Übersicht der FBs im Netzwerk

Bausteinname	Typ	Parameter	Ereignisverbindungen	Adapter-/Datenverbindungen
DigitalInput_I1	logiBUS::io::DI::logiBUS_DI_1CH	_IX1 = TRUEInput = Input_I1	–	Adapter `IN` → ILOCK_AX.UP_IN
DigitalInput_I2	logiBUS::io::DI::logiBUS_DI_1CH	_IX1 = TRUEInput = Input_I2	–	Adapter `IN` → ILOCK_AX.DOWN_IN
DigitalInput_Reset	logiBUS::io::DI::logiBUS_DI_1CH	_IX1 = TRUEInput = Input_I3, BUTTON_SINGLE_CLICK = TRUE	Ereignisausgang `IND` → ILOCK_AX.EI_RESET	–
ILOCK_AX	logiBUS::Logical_Processing::interlock::ILOCK_CONFLICT_TRIP_AX	*(keine Parameter gesetzt)*	Ereigniseingang `EI_RESET` von DigitalInput_Reset	Adaptereingänge: `UP_IN` (von DigitalInput_I1), `DOWN_IN` (von DigitalInput_I2)Adapterausgänge: `UP_OUT` → DigitalOutput_Q1, `DOWN_OUT` → DigitalOutput_Q2, `TRIP_OUT` → Trip_Anzeige
DigitalOutput_Q1	logiBUS::io::DQ::logiBUS_DQ_1CH	_QX1 = TRUEOutput = Output_Q1	–	Adaptereingang `OUT` von ILOCK_AX.UP_OUT
DigitalOutput_Q2	logiBUS::io::DQ::logiBUS_DQ_1CH	_QX1 = TRUEOutput = Output_Q2	–	Adaptereingang `OUT` von ILOCK_AX.DOWN_OUT
Trip_Anzeige	logiBUS::io::DQ::logiBUS_DQ_1CH	_QX1 = TRUEOutput = Output_Q4	–	Adaptereingang `OUT` von ILOCK_AX.TRIP_OUT

Programmablauf und Verbindungen

1. **Eingänge**:
   - Die digitalen Eingänge *Input_I1* (über FB „DigitalInput_I1“) und *Input_I2* (über FB „DigitalInput_I2“) stellen die Adapterschnittstellen `UP_IN` bzw. `DOWN_IN` für den ILOCK-Baustein bereit.
   - Der Reset-Eingang *Input_I3* wird über den FB „DigitalInput_Reset“ als Ereignis ausgewertet (Einzelflanke, Parameter `BUTTON_SINGLE_CLICK`). Das Ereignis `IND` triggert den Reset-Eingang `EI_RESET` des ILOCK-Bausteins.
2. **Interlock-Logik**:
   - Der Baustein **ILOCK_CONFLICT_TRIP_AX** überwacht die beiden Eingänge und erkennt einen Konflikt (z. B. gleichzeitige Anforderung in beide Richtungen).
   - Im Normalbetrieb gibt er die Signale unverändert an die Ausgänge `UP_OUT` und `DOWN_OUT` weiter.
   - Bei einem Konflikt wird der Ausgang `TRIP_OUT` aktiv gesetzt und die Ausgänge `UP_OUT`/`DOWN_OUT` werden in einen definierten (gesperrten) Zustand gebracht.
3. **Ausgänge**:
   - Der Adapterausgang `UP_OUT` steuert den digitalen Ausgang *Output_Q1* (FB „DigitalOutput_Q1“).
   - Der Adapterausgang `DOWN_OUT` steuert *Output_Q2* (FB „DigitalOutput_Q2“).
   - Der Trip-Ausgang `TRIP_OUT` aktiviert *Output_Q4* (FB „Trip_Anzeige“).
4. **Reset-Verhalten**:
   - Solange ein Trip aktiv ist, bleibt der `TRIP_OUT` gesetzt. Durch Betätigen des Reset-Eingangs (Einzelflanke auf *Input_I3*) wird der ILOCK-Baustein zurückgesetzt und die Ausgänge schalten wieder in den Normalzustand.

Zusammenfassung

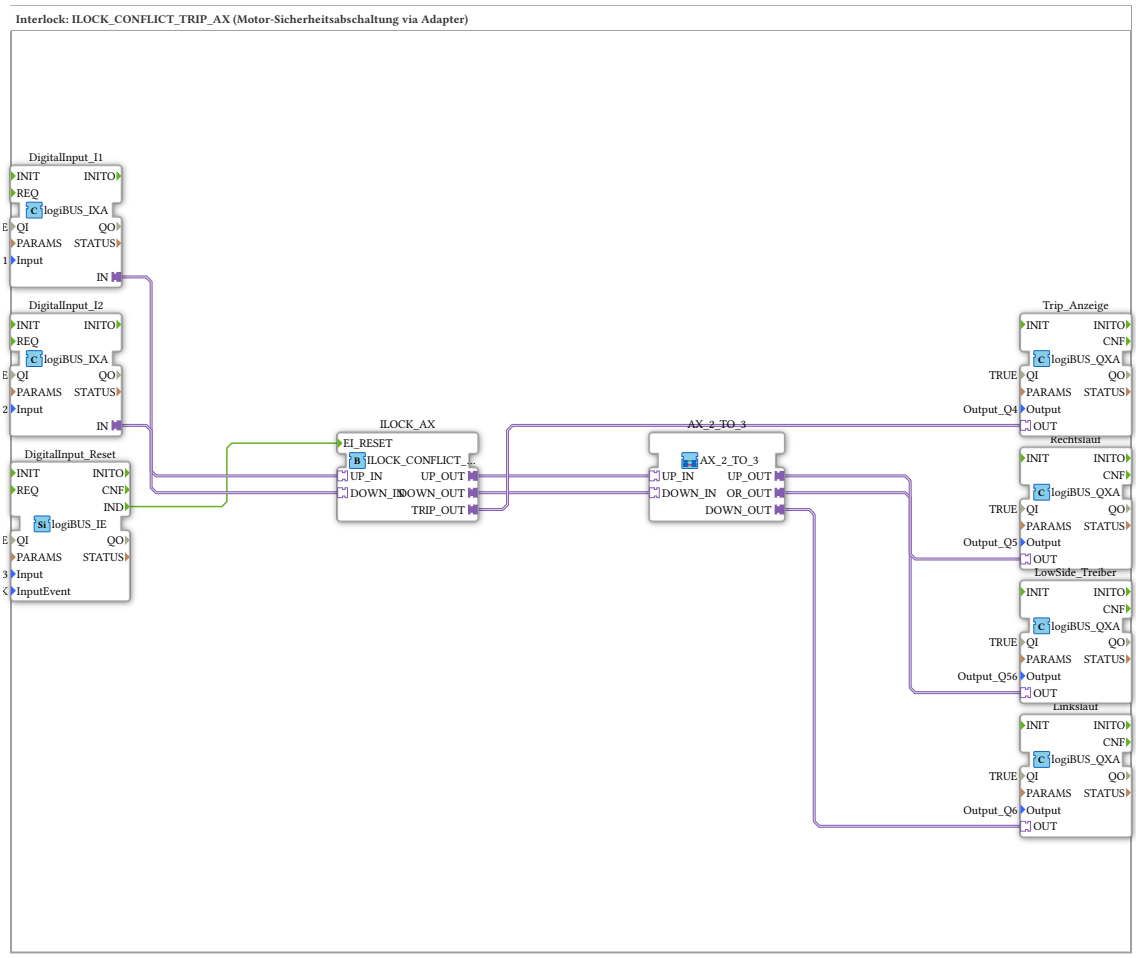
Die Übung **Uebung_204_AX** demonstriert den Einsatz eines Interlock-Bausteins mit integrierter Konflikterkennung und Trip-Auslösung über Adapter.
Sie verknüpft:

- zwei digitale Eingänge für Auf-/Ab-Signale,
- einen Reset-Eingang mit Flankenauswertung,
- den ILOCK-Baustein zur Konfliktüberwachung,
- drei digitale Ausgänge (Q1, Q2, Q4) zur Ansteuerung von Aktoren und Anzeige.

Das Beispiel eignet sich zur Einführung in die Interlock-Mechanismen der logiBUS-Bibliothek und zeigt, wie eine sichere Verriegelung mit automatischer Fehlerreaktion realisiert wird.

• – ○ • – ○ • – ○ •

2.1.2.479 Uebung_204b_AX: Interlock: ILOCK_CONFLICT_TRIP_AX (Motor-Sicherheitsabschaltung via Adapter)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert die Verwendung des Funktionsbausteins **ILOCK_CONFLICT_TRIP_AX** zur sicherheitsgerichteten Motorabschaltung. Es wird eine Interlock-Logik realisiert, bei der zwei gegensätzliche Anforderungen (z. B. Rechts- und Linkslauf) überwacht und bei einem Konflikt ein Trip ausgelöst wird. Die gesamte Steuerung erfolgt über Adapterverbindungen und einen zwischengeschalteten SubApp (AX_2_TO_3), der die Aufteilung der Signale auf die Ausgänge vornimmt.

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_I1** (Typ: `logiBUS::io::DI::logiBUS_IXA`)
  - Parameter: `QI = TRUE`, `Input = Input_I1`
  - Funktion: Digitaleingang für das erste Richtungssignal (z. B. Rechtslauf-Anforderung)
- **DigitalInput_I2** (Typ: `logiBUS::io::DI::logiBUS_IXA`)

  - Parameter: `QI` = `TRUE`, `Input` = `Input_I2`
  - Funktion: Digitaleingang für das zweite Richtungssignal (z. B. Linkslauf-Anforderung)
- **DigitalInput_Reset** (Typ: `logiBUS::io::DI::logiBUS_IE`)
  - Parameter: `QI` = `TRUE`, `Input` = `Input_I3`, `InputEvent` = `BUTTON_SINGLE_CLICK`
  - Funktion: Digitaleingang mit Ereignisunterstützung; dient als Reset-Taster für die Interlock-Einheit
- **ILOCK_AX** (Typ: `logiBUS::signalprocessing::interlock::ILOCK_CONFLICT_TRIP_AX`)
  - Parameter: keine
  - Funktion: Zentraler Interlock-Baustein. Überwacht die beiden Adaptereingänge **UP_IN** und **DOWN_IN** auf widersprüchliche Zustände. Bei einem Konflikt wird der Ausgang **TRIP_OUT** aktiviert. Bei konfliktfreien Zuständen werden **UP_OUT** und **DOWN_OUT** entsprechend durchgeschaltet.
- **Rechtslauf** (Typ: `logiBUS::io::DQ::logiBUS_QXA`)
  - Parameter: `QI` = `TRUE`, `Output` = `Output_Q5`
  - Funktion: Digitalausgang für Rechtslauf-Steuerung
- **Linkslauf** (Typ: `logiBUS::io::DQ::logiBUS_QXA`)
  - Parameter: `QI` = `TRUE`, `Output` = `Output_Q6`
  - Funktion: Digitalausgang für Linkslauf-Steuerung
- **LowSide_Treiber** (Typ: `logiBUS::io::DQ::logiBUS_QXA`)
  - Parameter: `QI` = `TRUE`, `Output` = `Output_Q56`
  - Funktion: Digitalausgang für einen gemeinsamen Low-Side-Treiber (z. B. Freigabe einer Motor-Endstufe)
- **Trip_Anzeige** (Typ: `logiBUS::io::DQ::logiBUS_QXA`)
  - Parameter: `QI` = `TRUE`, `Output` = `Output_Q4`
  - Funktion: Digitalausgang zur Anzeige eines ausgelösten Trips

Sub-Bausteine: AX_2_TO_3

- **Typ**: `MyLib::sys::AX_2_TO_3`
- **Verwendete interne FBs**: Keine Angabe – die interne Implementierung ist in dieser Übung nicht näher definiert und wird als Bibliotheksbaustein vorausgesetzt.
- **Funktionsweise**: Der SubApp dient als Adapter, der die beiden Adaptereingänge **UP_IN** und **DOWN_IN** auf drei separate Ausgänge verteilt: **UP_OUT**, **DOWN_OUT** und **OR_OUT**. Die genaue Logik ist abhängig von der internen Realisierung; in diesem Aufbau werden damit die Ausgänge für Rechtslauf, Linkslauf und den Low-Side-Treiber angesteuert.

Programmablauf und Verbindungen

1. **Eingangssignale**: Die digitalen Eingänge **I1** und **I2** (über `DigitalInput_I1` und `DigitalInput_I2`) werden als Adaptersignale an den Interlock-Baustein `ILOCK_AX` übergeben (Verbindungen: `DigitalInput_I1.IN` ⟶ `ILOCK_AX.UP_IN`, `DigitalInput_I2.IN` ⟶ `ILOCK_AX.DOWN_IN`).
2. **Interlock-Logik**: `ILOCK_AX` prüft auf widersprüchliche Anforderungen. Falls beide Eingänge gleichzeitig aktiv sind, wird der Trip-Ausgang **TRIP_OUT** gesetzt. Bei konfliktfreien Zuständen werden die Signale auf **UP_OUT** und **DOWN_OUT** durchgeschleift.
3. **Signalverteilung**: Die Ausgänge von `ILOCK_AX` werden an den SubApp `AX_2_TO_3` weitergeleitet:
   - `ILOCK_AX.UP_OUT` ⟶ `AX_2_TO_3.UP_IN`
   - `ILOCK_AX.DOWN_OUT` ⟶ `AX_2_TO_3.DOWN_IN`
   - `ILOCK_AX.TRIP_OUT` ⟶ `Trip_Anzeige.OUT`
4. **Endstufenansteuerung**: Der SubApp `AX_2_TO_3` teilt die Signale auf die finalen Ausgänge auf:
   - `AX_2_TO_3.UP_OUT` ⟶ `Rechtslauf.OUT`
   - `AX_2_TO_3.DOWN_OUT` ⟶ `Linkslauf.OUT`
   - `AX_2_TO_3.OR_OUT` ⟶ `LowSide_Treiber.OUT`

5. **Reset-Funktion**: Der digitale Eingang **I3** (über `DigitalInput_Reset`) wird mit einem Tastendruck-Ereignis (`BUTTON_SINGLE_CLICK`) an den Ereigniseingang **EI_RESET** von `ILOCK_AX` angeschlossen, um einen ausgelösten Trip zurückzusetzen.

Hinweise zur Übung:

- Schwierigkeitsgrad: Mittel
- Lernziele: Verständnis von Interlock-Mechanismen, Adapter-basierter Kommunikation, Signalweiterleitung über SubApp
- Voraussetzung: Grundkenntnisse in 4diac-IDE und IEC 61499

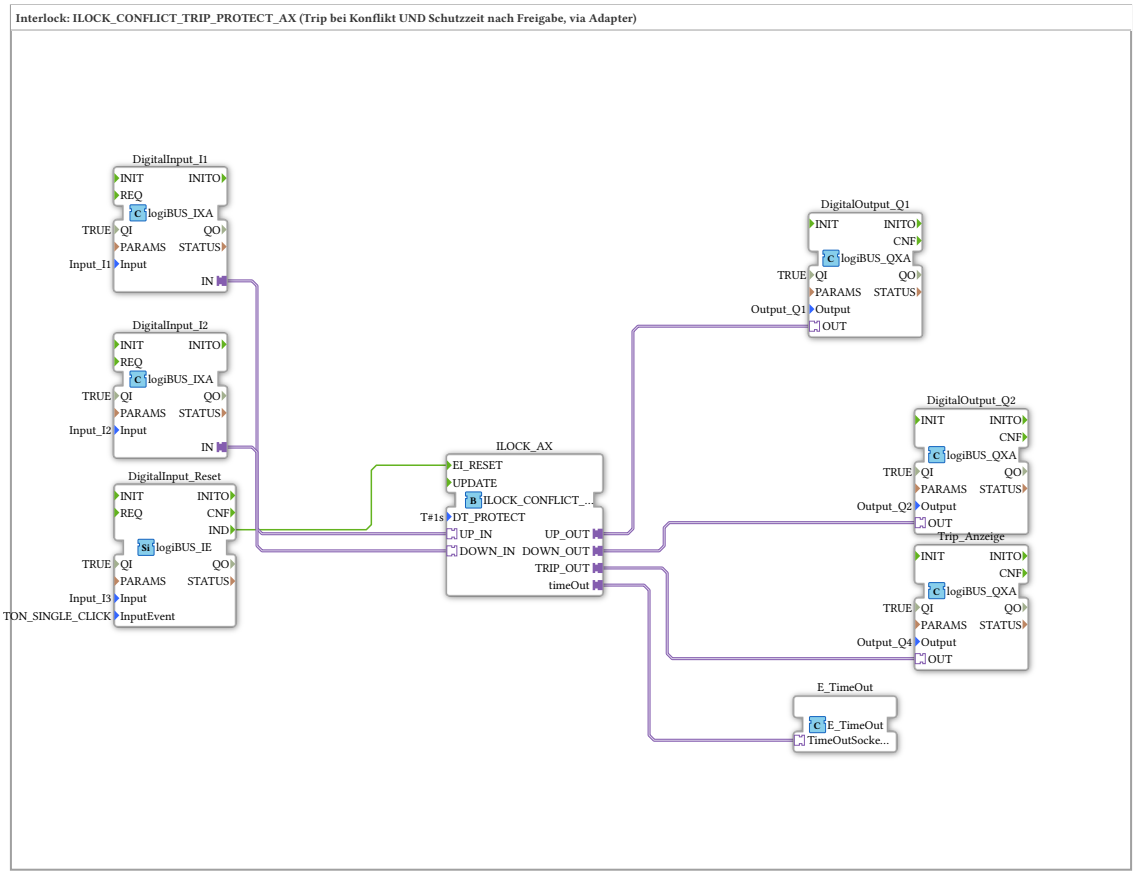
Zusammenfassung

Die Übung vermittelt den Aufbau einer sicherheitsgerichteten Motorsteuerung unter Verwendung des Interlock-Bausteins `ILOCK_CONFLICT_TRIP_AX`. Durch die strukturierte Aufteilung der Signale mittels eines SubApp (`AX_2_TO_3`) werden die Ausgänge für Rechtslauf, Linkslauf, Low-Side-Treiber und Trip-Anzeige realisiert. Ein Reset-Eingang erlaubt das Zurücksetzen der Sicherheitsabschaltung.

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

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

2.1.2.480 Uebung_204c_AX: Interlock ILOCK_CONFLICT_TRIP_PROTECT_AX (Trip bei Konflikt UND Schutzzeit nach Freigabe, via Adapter)



• — ○ • — ○ • — ○ •

Einleitung

Diese Übung erweitert `Uebung_204_AX` (die den einfachen `ILOCK_CONFLICT_TRIP_AX` ohne Schutzzeit verwendet) um eine zusätzliche Schutzzeit: Nach Freigabe des aktiven Eingangs wartet die Verriegelung `DT_PROTECT` (hier 1 Sekunde) ab, bevor sie eine neue Richtung übernimmt. Diese Wartezeit gilt ausschließlich für die Übernahme einer neuen Richtung – ein Konflikt, der auftritt, während der erste Eingang

noch aktiv gehalten wird (also vor dessen Freigabe), löst weiterhin sofort einen TRIP aus, unabhängig von DT_PROTECT (unverändert gegenüber ILOCK_CONFLICT_TRIP). Ist nach Ablauf von DT_PROTECT jedoch immer noch (oder wieder) der jeweils andere Eingang aktiv, wird bei der dann fälligen Neubewertung statt der neuen Richtung erneut ein TRIP ausgelöst. Es handelt sich um dieselbe Schutzzeit-Logik wie bei ILOCK_SWITCH_PROTECT_AX (Uebung_205_AX), hier kombiniert mit der Konflikterkennung (TRIP) aus Uebung_204_AX.

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_I1**, **DigitalInput_I2**: logiBUS Digitaleingänge (Typ: logiBUS::io::DI::logiBUS_IXA)
  - **Parameter**: QI = TRUE, Input = Input_I1 bzw. Input_I2
  - **Erklärung**: Liefern die beiden sich gegenseitig ausschließenden Anforderungssignale (z. B. Auf/Ab) als AX-Adaptersignale an die Verriegelung.
- **DigitalInput_Reset**: logiBUS Ereigniseingang (Typ: logiBUS::io::DI::logiBUS_IE)
  - **Parameter**: QI = TRUE, Input = Input_I3, InputEvent = BUTTON_SINGLE_CLICK
  - **Erklärung**: Löst bei einem Tasterklick das Rücksetzen (EI_RESET) der Verriegelung aus, nachdem ein Konflikt (TRIP) aufgetreten ist.
- **ILOCK_AX**: Interlock-Baustein (Typ: logiBUS::signalprocessing::interlock::ILOCK_CONFLICT_TRIP_PROTECT_AX)
  - **Parameter**: DT_PROTECT = T#1s
  - **Erklärung**: Verriegelt die beiden Eingänge UP_IN/DOWN_IN gegenseitig. Sind beide gleichzeitig aktiv, während eine Richtung noch aktiv gehalten wird, erkennt der Baustein sofort einen Konflikt (TRIP) und setzt TRIP_OUT – unabhängig von DT_PROTECT. Nach Freigabe des zuvor aktiven Eingangs wartet er zusätzlich DT_PROTECT ab und bewertet danach die dann aktuelle Eingangslage neu: liegt nur noch eine Richtung an, wird sie übernommen; sind (wieder) beide Eingänge aktiv, wird stattdessen erneut ein TRIP ausgelöst.
- **DigitalOutput_Q1**: logiBUS Digitalausgang (Typ: logiBUS::io::DQ::logiBUS_QXA)
  - **Parameter**: QI = TRUE, Output = Output_Q1
  - **Erklärung**: Gibt das freigegebene UP_OUT-Signal an die Peripherie weiter.
- **DigitalOutput_Q2**: logiBUS Digitalausgang (Typ: logiBUS::io::DQ::logiBUS_QXA)
  - **Parameter**: QI = TRUE, Output = Output_Q2
  - **Erklärung**: Gibt das freigegebene DOWN_OUT-Signal an die Peripherie weiter.
- **Trip_Anzeige**: logiBUS Digitalausgang (Typ: logiBUS::io::DQ::logiBUS_QXA)
  - **Parameter**: QI = TRUE, Output = Output_Q4
  - **Erklärung**: Zeigt den Konfliktzustand (TRIP_OUT) an, z. B. über eine Meldeleuchte.
- **E_TimeOut**: Zeitgeber-Ereignisquelle (Typ: iec61499::events::E_TimeOut)
  - **Parameter**: keine
  - **Erklärung**: Empfängt über die Adapterverbindung timeOut das von ILOCK_AX gestartete Zeitsignal (DT_PROTECT/START), verarbeitet es und meldet den Ablauf über timeOut.TimeOut zurück.

Sub-Bausteine: keine

Die Übung verwendet keine weiteren Unterbausteine, alle FBs sind direkt auf der obersten Ebene der SubApp angeordnet.

Programmablauf und Verbindungen

1. DigitalInput_I1.IN → ILOCK_AX.UP_IN und DigitalInput_I2.IN → ILOCK_AX.DOWN_IN: Die beiden Anforderungssignale gelangen über AdapterConnections in die Verriegelung.
2. DigitalInput_Reset.IND → ILOCK_AX.EI_RESET: Ein Klick auf den Reset-Taster setzt einen ausgelösten Konflikt zurück.
3. ILOCK_AX.timeOut → E_TimeOut.TimeOutSocket: ILOCK_AX setzt darüber DT_PROTECT und startet den Timer; E_TimeOut verarbeitet das Zeitsignal und meldet den Ablauf zurück.
4. **Normalbetrieb**: Ist nur ein Eingang aktiv, wird UP_OUT bzw. DOWN_OUT freigegeben und über Output_Q1/Output_Q2 ausgegeben.
5. **Konfliktfall (TRIP)**: Sind UP_IN und DOWN_IN gleichzeitig aktiv, setzt ILOCK_AX.TRIP_OUT → Trip_Anzeige.OUT und blockiert beide Ausgänge, bis über DigitalInput_Reset zurückgesetzt wird.
6. **Schutzzeit**: Wird der aktive Eingang losgelassen, wartet ILOCK_AX DT_PROTECT (1 s) ab und bewertet danach die dann aktuelle Eingangslage neu: liegt nur eine Richtung an, wird sie übernommen; sind beide Eingänge aktiv (weil der andere währenddessen aktiviert wurde und noch ansteht), wird stattdessen ein TRIP ausgelöst. Ein Konflikt, der auftritt, während der erste Eingang noch gehalten wird (also vor dessen Freigabe), löst dagegen weiterhin sofort TRIP aus, ohne auf DT_PROTECT zu warten.

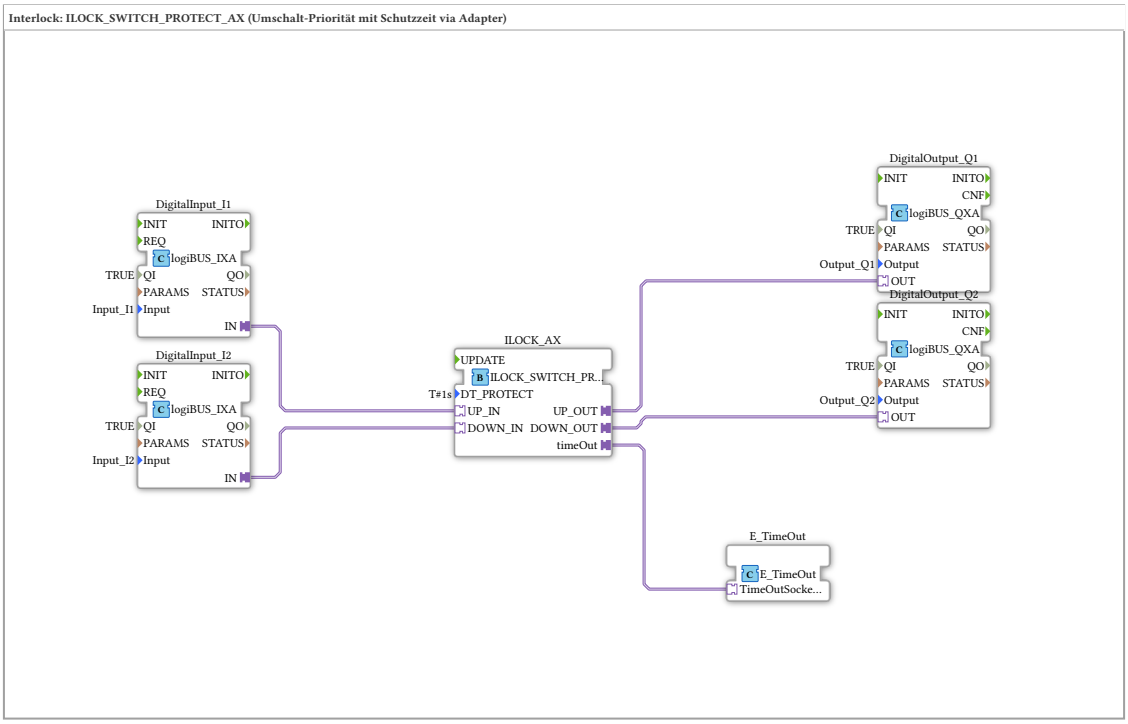
Zusammenfassung

Die Übung 204c_AX kombiniert die Konflikterkennung aus `Uebung_204_AX` mit einer Schutzzeit nach Freigabe des aktiven Eingangs, wie sie `Uebung_205_AX` für einfache Richtungswechsel einführt. `ILOCK_CONFLICT_TRIP_PROTECT_AX` verhindert dadurch nicht nur gleichzeitige, widersprüchliche Anforderungen, sondern auch ein zu schnelles Umschalten der Richtung unmittelbar nach Freigabe – eine typische Anforderung an robuste Verriegelungslogik in der Automatisierungstechnik.

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

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

2.1.2.481 Uebung_205_AX: Interlock: ILOCK_SWITCH_PROTECT_AX (Umschalt-Priorität mit Schutzzeit via Adapter)



• – ○ • – ○ • – ○ •

Einleitung

Die Übung **Uebung_205_AX** realisiert eine sichere Umschaltsteuerung mit Priorität und Schutzzeit. Zwei digitale Eingänge (I1, I2) steuern über einen Interlock-Baustein zwei digitale Ausgänge (Q1, Q2). Der Interlock verhindert gleichzeitige Aktivierungen und erzwingt eine Schutzzeit von 1 s zwischen Umschaltvorgängen.
Ein zusätzlicher E_TimeOut-Baustein dient der Überwachung des Timeout-Signals.

Die Übung ist als SubAppType modelliert und verwendet logiBUS-Adapter für die Ein-/Ausgabeanbindung.

Verwendete Funktionsbausteine (FBs)

Die SubApp enthält folgende Funktionsbausteine:

Sub-Bausteine: DigitalInput_I1 & DigitalInput_I2 (logiBUS_IXA)

- **Typ**: `logiBUS::io::DI::logiBUS_IXA`
- **Verwendete interne FBs**: keine (einfacher Adapterbaustein)
- **Parameter**:
  - `QI` = TRUE
  - `Input` = `Input_I1` bzw. `Input_I2`

- **Funktionsweise**:
  Liest den jeweiligen digitalen Eingang (I1/I2) ein und stellt den Signalzustand am Ausgang `IN` bereit.

Sub-Baustein: ILOCK_AX (ILOCK_SWITCH_PROTECT_AX)

- **Typ**: `logiBUS::signalprocessing::interlock::ILOCK_SWITCH_PROTECT_AX`
- **Verwendete interne FBs**: keine (komplexer, vorgefertigter Baustein)
- **Parameter**:
  - `DT_PROTECT` = T#1s (Schutzzeit 1 Sekunde)
- **Ereignis-/Datenschnittstellen**:
  - **Adapter-Eingänge**: `UP_IN`, `DOWN_IN`
  - **Adapter-Ausgänge**: `UP_OUT`, `DOWN_OUT`, `timeOut`
- **Funktionsweise**:
  Realisiert eine **umschaltpriorisierte Verriegelung** mit Schutzzeit.
  – Bei aktivem `UP_IN` wird der Ausgang `UP_OUT` gesetzt, während `DOWN_OUT` für die Schutzzeit gesperrt bleibt.
  – Bei aktivem `DOWN_IN` analog in die andere Richtung.
  – Das Signal `timeOut` wird aktiv, wenn die Schutzzeit überschritten wird (z. B. Blockade oder Überlast).

Sub-Bausteine: DigitalOutput_Q1 & DigitalOutput_Q2 (logiBUS_QXA)

- **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
- **Verwendete interne FBs**: keine (einfacher Adapterbaustein)
- **Parameter**:
  - `QI` = TRUE
  - `Output` = `Output_Q1` bzw. `Output_Q2`
- **Funktionsweise**:
  Setzt den jeweiligen digitalen Ausgang (Q1/Q2) gemäß dem am Eingang `OUT` anliegenden Signal.

Sub-Baustein: E_TimeOut (E_TimeOut)

- **Typ**: `iec61499::events::E_TimeOut`
- **Verwendete interne FBs**: keine
- **Parameter**: keine (Standardeinstellungen)
- **Funktionsweise**:
  Erzeugt ein Ereignis an seinem Ausgang, sobald eine Zeitverzögerung abgelaufen ist.
  In dieser Übung ist der Eingang `TimeOutSocket` mit dem `timeOut`-Ausgang des Interlock-Bausteins verbunden – zur weiteren Verarbeitung (z. B. Alarmierung).

Programmablauf und Verbindungen

1. **DigitalInput_I1** und **DigitalInput_I2** lesen die binären Signale von `Input_I1` bzw. `Input_I2` ein.
2. Die **Adapter-Ausgänge `IN`** dieser Bausteine sind mit den **Adapter-Eingängen `UP_IN`** bzw. **`DOWN_IN`** des Interlock-Bausteins verbunden.
3. **ILOCK_AX** wertet die Eingänge aus, wendet die Schutzzeit (`DT_PROTECT = T#1s`) an und steuert seine Ausgänge:
   - `UP_OUT` → verbunden mit dem **OUT-Eingang** von **DigitalOutput_Q1**
   - `DOWN_OUT` → verbunden mit dem **OUT-Eingang** von **DigitalOutput_Q2**
   - `timeOut` → verbunden mit dem **TimeOutSocket** des **E_TimeOut**-Bausteins
4. **DigitalOutput_Q1** und **DigitalOutput_Q2** setzen die entsprechenden physischen Ausgänge (`Output_Q1`, `Output_Q2`).
5. Der **E_TimeOut**-Baustein kann optional für eine Zeitüberwachung genutzt werden (z. B. um einen Alarm bei längerem Timeout auszulösen).

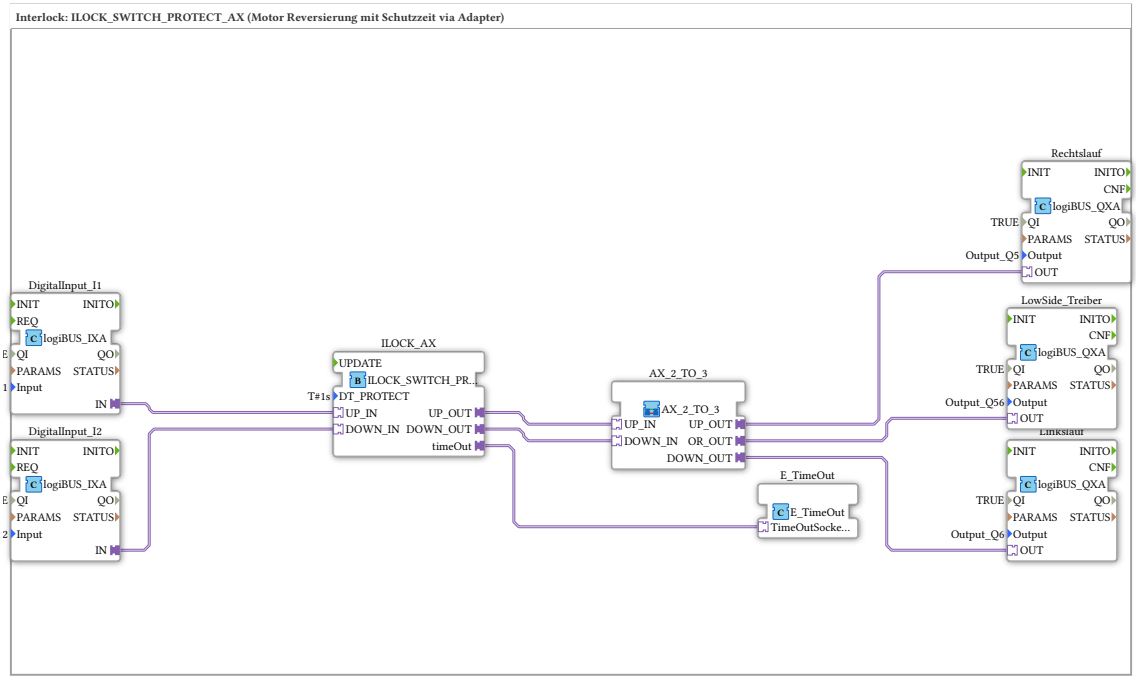
Lernziel:
Diese Übung vermittelt den Umgang mit Interlock-Bausteinen zur Realisierung von **Prioritätssteuerungen** und **Schutzzeiten** in Steuerungsanwendungen. Sie zeigt die Kopplung von digitalen Ein-/Ausgängen über Adapter sowie die Überwachung von Timeout-Signalen.
Schwierigkeitsgrad: Mittel
Vorkenntnisse: Grundlagen der IEC 61499-Modellierung, Umgang mit logiBUS-Adaptern.

Zusammenfassung

Die Übung **Uebung_205_AX** demonstriert den Einsatz des Interlock-Bausteins `ILOCK_SWITCH_PROTECT_AX` in einer SubApp.

Durch die Schutzzeit von einer Sekunde wird eine sichere und priorisierte Umschaltung zwischen zwei digitalen Eingängen gewährleistet.
Die Ein-/Ausgabe erfolgt über logiBUS-Adapter, der integrierte E_TimeOut-Baustein erlaubt eine einfache Überwachung des Timeout-Zustands.
Die Übung eignet sich als Grundlage für sicherheitsgerichtete Steuerungen wie Motorumschaltungen oder Ventilsteuerungen.

2.1.2.482 Uebung_205b_AX: Interlock: ILOCK_SWITCH_PROTECT_AX (Motor Reversierung mit Schutzzeit via Adapter)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung behandelt die Motorreversierung mit Schutzzeit unter Verwendung eines Interlock-Bausteins (ILOCK_SWITCH_PROTECT_AX). Ziel ist es, einen Motor über zwei Eingänge (Aufwärts/Abwärts) zu steuern, wobei eine festgelegte Schutzzeit (hier 1 Sekunde) das gleichzeitige Einschalten beider Richtungen verhindert und einen zu schnellen Richtungswechsel blockiert. Die Steuerung erfolgt über Adapter-Schnittstellen, was die Wiederverwendbarkeit des Subapplikationstyps erhöht.

Schwierigkeitsgrad: Fortgeschritten Vorkenntnisse: Grundlagen der 4diac-IDE, Umgang mit Ein-/Ausgangsbausteinen, Verständnis von Interlock-Logiken.

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_I1** (logiBUS::io::DI::logiBUS_IXA)
  - Parameter: QI = TRUE, Input = Input_I1
  - Ereignisausgang/-eingang: —
  - Datenausgang/-eingang: IN (Adapter-Schnittstelle)
- **DigitalInput_I2** (logiBUS::io::DI::logiBUS_IXA)
  - Parameter: QI = TRUE, Input = Input_I2
  - Ereignisausgang/-eingang: —
  - Datenausgang/-eingang: IN (Adapter-Schnittstelle)
- **ILOCK_AX** (logiBUS::signalprocessing::interlock::ILOCK_SWITCH_PROTECT_AX)
  - Parameter: DT_PROTECT = T#1s
  - Ereignisausgang/-eingang: timeOut (Ereignisausgang)
  - Datenausgang/-eingang: UP_IN, DOWN_IN (Eingänge); UP_OUT, DOWN_OUT (Ausgänge)

- **Rechtslauf** (logiBUS::io::DQ::logiBUS_QXA)
  - Parameter: QI = TRUE, Output = Output_Q5
  - Ereignisausgang/-eingang: —
  - Datenausgang/-eingang: OUT (Adapter-Schnittstelle)
- **Linkslauf** (logiBUS::io::DQ::logiBUS_QXA)
  - Parameter: QI = TRUE, Output = Output_Q6
  - Ereignisausgang/-eingang: —
  - Datenausgang/-eingang: OUT (Adapter-Schnittstelle)
- **LowSide_Treiber** (logiBUS::io::DQ::logiBUS_QXA)
  - Parameter: QI = TRUE, Output = Output_Q56
  - Ereignisausgang/-eingang: —
  - Datenausgang/-eingang: OUT (Adapter-Schnittstelle)
- **E_TimeOut** (iec61499::events::E_TimeOut)
  - Parameter: keine (Standard-Timeout-Baustein)
  - Ereignisausgang/-eingang: TimeOutSocket (Ereigniseingang, verbunden mit ILOCK_AX.timeOut)
  - Datenausgang/-eingang: —

Sub-Bausteine: AX_2_TO_3

- **Typ**: MyLib::sys::AX_2_TO_3
- **Verwendete interne FBs**: (keine detaillierten Informationen in der Übung enthalten)
- **Funktionsweise**: Dieser Sub-Baustein dient der Weiterleitung und logischen Verknüpfung der Auf- und Abwärtssignale.
  - **UP_IN** und **DOWN_IN** werden als Eingänge empfangen.
  - **UP_OUT** gibt das Signal von UP_IN unverändert weiter.
  - **DOWN_OUT** gibt das Signal von DOWN_IN unverändert weiter.
  - **OR_OUT** ist eine ODER-Verknüpfung der beiden Eingänge (UP_IN ODER DOWN_IN). Dieses Signal wird verwendet, um den LowSide-Treiber zu aktivieren, sobald eine der beiden Richtungen angefordert wird.

  Damit werden die getrennten Richtungssignale auf zwei Ausgänge verteilt, gleichzeitig aber ein gemeinsames Signal für die Low-Side-Ansteuerung erzeugt.

Programmablauf und Verbindungen

1. **Eingangssignale** Die digitalen Eingänge `Input_I1` und `Input_I2` werden über die Bausteine `DigitalInput_I1` und `DigitalInput_I2` eingelesen. Diese stellen die Anforderungen `UP_IN` und `DOWN_IN` an den Interlock-Baustein `ILOCK_AX` bereit.
2. **Interlock-Logik** `ILOCK_AX` wertet die beiden Anforderungen aus.
   - Bei einer aktiven Anforderung (z.B. `UP_IN`) wird der entsprechende Ausgang (`UP_OUT`) aktiviert, sofern nicht gleichzeitig die Gegenrichtung anliegt.
   - Die Schutzzeit `DT_PROTECT = 1s` verhindert, dass nach einem Richtungswechsel sofort die andere Richtung geschaltet werden kann.
   - Wenn die Schutzzeit aktiv ist und eine Anforderung für die Gegenrichtung kommt, wird der Ausgang blockiert und der `timeOut`-Ereignisausgang getriggert.
3. **Zeitüberwachung** Der Ereignisausgang `timeOut` von `ILOCK_AX` ist mit dem `E_TimeOut`-Baustein verbunden. Dieser kann z.B. für weitere Verarbeitung oder Visualisierung genutzt werden (hier nicht weiter ausgeführt).
4. **Signalverteilung via AX_2_TO_3** Die Ausgänge `UP_OUT` und `DOWN_OUT` von `ILOCK_AX` werden an die SubApp `AX_2_TO_3` übergeben.
   - `UP_OUT` ⟶ `AX_2_TO_3.UP_IN`
   - `DOWN_OUT` ⟶ `AX_2_TO_3.DOWN_IN` Der Sub-Baustein leitet die Signale getrennt an die Ausgangsbausteine `Rechtslauf` (Q5) und `Linkslauf` (Q6) weiter. Das ODER-Signal `OR_OUT` aktiviert den `LowSide_Treiber` (Q56), der die gemeinsame Low-Side-Versorgung für den Motor schaltet.
5. **Ausgangsbausteine**

- `Rechtslauf` (Output_Q5): Steuert das Relais für Rechtslauf.
- `Linkslauf` (Output_Q6): Steuert das Relais für Linkslauf.
- `LowSide_Treiber` (Output_Q56): Schaltet die Low-Side-Spannung – notwendig, sobald eine der beiden Richtungen aktiv ist.

Die gesamte Logik ist als wiederverwendbare SubApp gekapselt und kann in übergeordneten Steuerungsprojekten eingebunden werden.

Zusammenfassung

Die Übung **Uebung_205b_AX** vermittelt die Umsetzung einer motorischen Reversiersteuerung mit Interlock und Schutzzeit. Durch die Verwendung des spezialisierten Bausteins `ILOCK_SWITCH_PROTECT_AX` wird eine sichere Richtungsumkehr gewährleistet. Die Adapter-basierte Kommunikation zwischen den Bausteinen ermöglicht eine flexible und modulare Struktur. Die SubApp `AX_2_TO_3` übernimmt die Aufteilung der Richtungssignale und die Erzeugung eines gemeinsamen Low-Side-Signals. Diese Übung eignet sich zur Vertiefung des Verständnisses von Interlock-Logiken sowie der Arbeit mit Adaptern in 4diac.

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

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

2.1.2.483 Uebung_206_AX: Interlock: ILOCK_T_FF_AX (Drei gegenseitig verriegelte Toggle-Flip-Flops via Adapter-Kette)

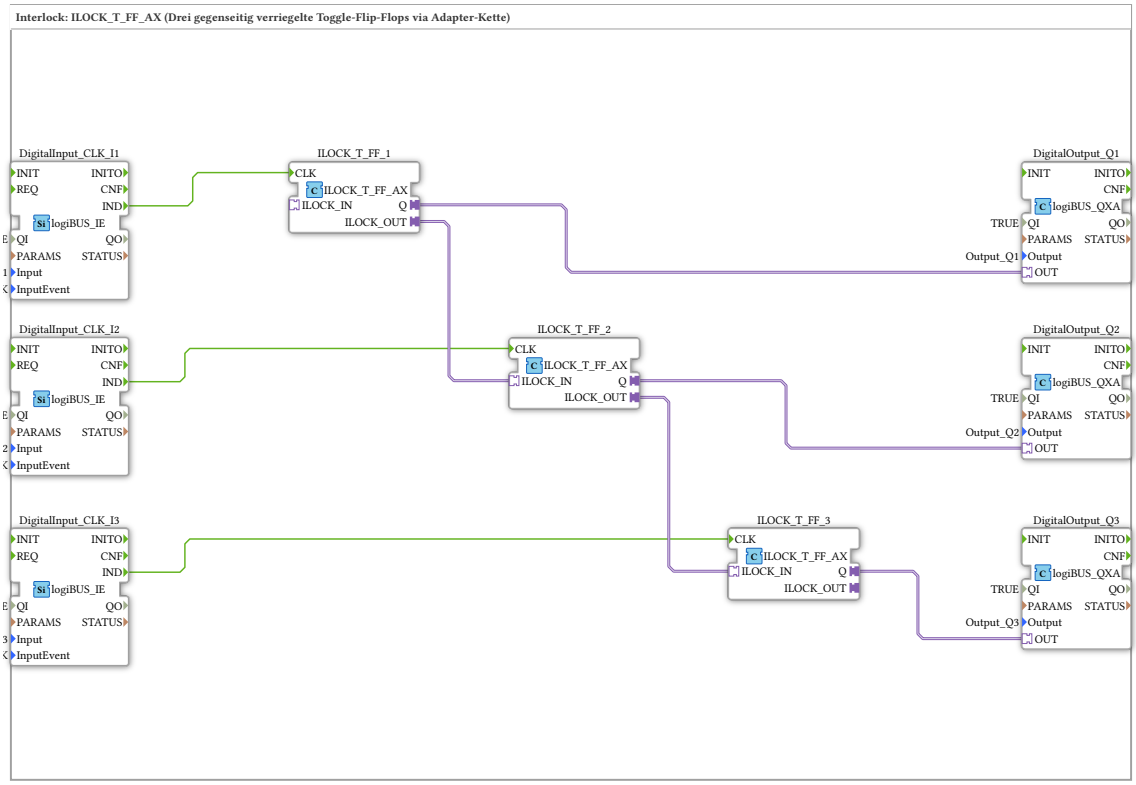


Bild der Übung nicht verfügbar

• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert die Umsetzung eines **Interlocks** (gegenseitige Verriegelung) mit drei Toggle-Flip-Flops (T-FF). Jedes Flip-Flop wird durch einen Taster (Single-Click) umgeschaltet. Die Besonderheit: Über eine bidirektionale Adapter-Kette wird sichergestellt, dass immer nur genau ein Ausgang aktiv sein kann – sobald ein Flip-Flop gesetzt wird, werden die anderen automatisch zurückgesetzt.

Dadurch eignet sich die Schaltung für sicherheitskritische Anwendungen, z. B. zur exklusiven Ansteuerung von Aktoren.

Verwendete Funktionsbausteine (FBs)

Bausteinname	Typ	Beschreibung
DigitalInput_CLK_I1	logiBUS::io::DI::logiBUS_IE	Digitaler Eingang für Taster an Kanal I1
DigitalInput_CLK_I2	logiBUS::io::DI::logiBUS_IE	Digitaler Eingang für Taster an Kanal I2
DigitalInput_CLK_I3	logiBUS::io::DI::logiBUS_IE	Digitaler Eingang für Taster an Kanal I3
ILOCK_T_FF_1	logiBUS::signalprocessing::interlock::ILOCK_T_FF_AX	Gegenseitig verriegeltes Toggle-Flip-Flop (zentraler Baustein)
ILOCK_T_FF_2	logiBUS::signalprocessing::interlock::ILOCK_T_FF_AX	Gleicher Typ wie ILOCK_T_FF_1
ILOCK_T_FF_3	logiBUS::signalprocessing::interlock::ILOCK_T_FF_AX	Gleicher Typ wie ILOCK_T_FF_1
DigitalOutput_Q1	logiBUS::io::DQ::logiBUS_QXA	Digitaler Ausgang an Kanal Q1
DigitalOutput_Q2	logiBUS::io::DQ::logiBUS_QXA	Digitaler Ausgang an Kanal Q2
DigitalOutput_Q3	logiBUS::io::DQ::logiBUS_QXA	Digitaler Ausgang an Kanal Q3

Sub-Baustein: ILOCK_T_FF_AX

- **Typ**: `logiBUS::signalprocessing::interlock::ILOCK_T_FF_AX` (Bibliotheksbaustein)
- **Verwendete interne FBs**: *keine detaillierten Informationen öffentlich* – der Baustein ist als vorgefertigter Logikbaustein aus der Bibliothek eingebunden.
- **Funktionsweise**:
  - Der Baustein realisiert ein **Toggle-Flip-Flop** (T-FF): Bei jedem positiven Flanke am Takteingang `CLK` wechselt der Ausgang `Q` seinen Zustand (von FALSE auf TRUE oder umgekehrt).
  - Zusätzlich besitzt er eine **Adapter-Schnittstelle** (`ILOCK_IN` / `ILOCK_OUT`), über die eine Verriegelungskette aufgebaut wird. Sobald der eigene Ausgang `Q` auf TRUE gesetzt wird, sendet der Baustein ein Sperrsignal über `ILOCK_OUT`. Empfangt er ein Sperrsignal von einem vorherigen Baustein über `ILOCK_IN`, wird der eigene Ausgang sofort auf FALSE zurückgesetzt (falls aktiv). Dadurch kann immer nur ein Flip-Flop in der Kette aktiv sein.
- **Parameter**:
  - Keine benutzerspezifischen Parameter (alle Standardwerte aus der Bibliothek).
- **Ereignisschnittstellen**:
  - **Eingang**: `CLK` – Ereignis (positive Flanke) zum Umschalten des Ausgangs.
  - **Ausgang**: *keine eigenen Ereignisausgänge* (die Ausgangsdaten werden direkt über Adapter weitergegeben).
- **Datenschnittstellen**:
  - **Datenausgang**: `Q` (BOOL) – der aktuelle Zustand des Flip-Flops.
- **Adapterschnittstellen**:
  - **Plug**: `ILOCK_IN` – Adaptereingang für das Sperrsignal des Vorgängers.
  - **Socket**: `ILOCK_OUT` – Adapterausgang zum Sperren des Nachfolgers.

Programmablauf und Verbindungen

1. **Eingangsereignisse** Die drei Digital-Eingänge (`DigitalInput_CLK_Ix`) wandeln Taster-Signale (Single-Click-Ereignis) in Ereignisse am Ausgang `IND` um. Diese werden direkt mit dem `CLK`-Eingang des jeweiligen `ILOCK_T_FF_Ax`-Bausteins verbunden.
2. **Interlock-Kette** Die drei Flip-Flops werden über ihre Adapter-Schnittstellen in einer Kette verbunden:
   - `ILOCK_T_FF_1.ILOCK_OUT` ⟶ `ILOCK_T_FF_2.ILOCK_IN`
   - `ILOCK_T_FF_2.ILOCK_OUT` ⟶ `ILOCK_T_FF_3.ILOCK_IN`

- (Der Ausgang von `ILOCK_T_FF_3.ILOCK_OUT` bleibt ungenutzt; die Kette ist an dieser Stelle offen.)

  Durch diese Verkettung wird sichergestellt, dass bei Setzen eines Flip-Flops (z. B. Nr. 1) der Nachfolger (Nr. 2) ein Sperrsignal erhält, das wiederum an Nr. 3 weitergegeben wird. Sobald in der Kette ein aktives Sperrsignal anliegt, schaltet der betroffene Baustein seinen Ausgang `Q` sofort aus.

3. **Ausgangsansteuerung** Die Ausgänge `Q` der drei Flip-Flops sind mit den Digitalausgängen (`DigitalOutput_Q1` … `DigitalOutput_Q3`) verbunden. Diese geben den Zustand an die Hardware-Kanäle Q1, Q2 und Q3 weiter.

Lernziele:

- Verständnis des Interlock-Prinzips (gegenseitige Verriegelung) in der Automatisierungstechnik.
- Kennenlernen des Bibliotheksbausteins `ILOCK_T_FF_AX` und seiner Adapter-Schnittstelle.
- Anwendung von Ereignis- und Adapter-Verbindungen in 4diac.

Schwierigkeitsgrad: Mittel – Vorkenntnisse in IEC 61499, Grundlagen der Ereignissteuerung und Adapter-Handhabung werden empfohlen.

Hinweise zur Ausführung:

- Die Übung ist für den Einsatz mit einem logiBUS-System (z. B. Raspberry Pi mit I/O-Erweiterung) ausgelegt.
- Vor dem Start müssen die Hardware-Kanäle (Input_I1 … Input_I3, Output_Q1 … Output_Q3) korrekt an die realen Taster und Aktoren angeschlossen sein.
- Der Baustein `logiBUS_DI_Events::BUTTON_SINGLE_CLICK` wird als Ereignisquelle genutzt – ein einfacher Tastendruck erzeugt ein einzelnes Takt-Ereignis (Entprellung ist im Treiber integriert).

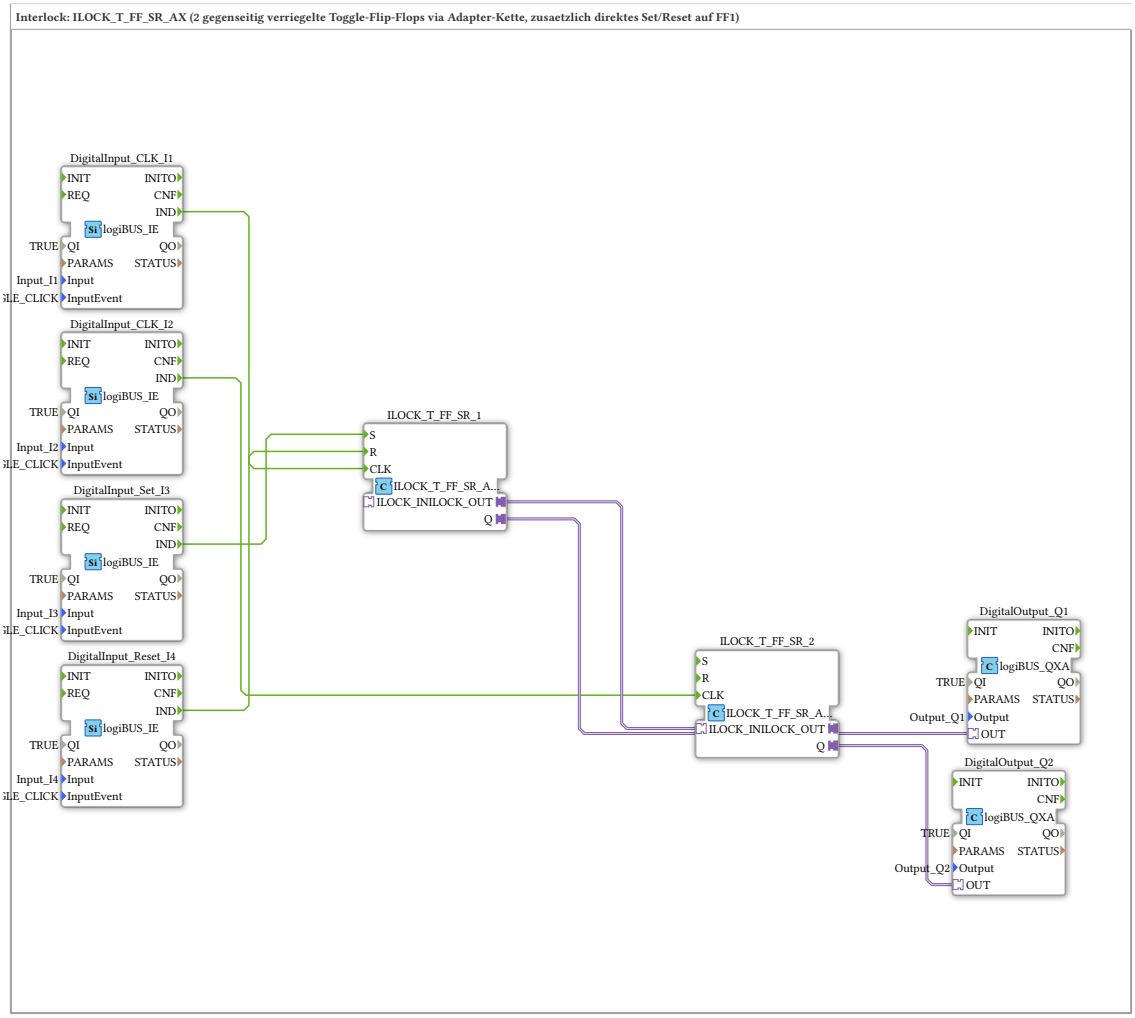
Zusammenfassung

Die Übung „Uebung_206_AX“ zeigt einen eleganten Weg, drei Toggle-Flip-Flops gegenseitig zu verriegeln, sodass stets nur ein Ausgang aktiv ist. Die Verwendung einer Adapter-Kette vereinfacht die Verkabelung und macht die Logik modular erweiterbar auf mehrere Stufen. Der Bibliotheksbaustein `ILOCK_T_FF_AX` kapselt die komplexe Verriegelungslogik und erlaubt eine klare, übersichtliche Netzwerktopologie.

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

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

2.1.2.484 Uebung_206b_AX: Interlock ILOCK_T_FF_SR_AX (2 gegenseitig verriegelte Toggle-Flip-Flops, zusätzlich direktes Set/Reset auf FF1)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung erweitert `Uebung_206_AX` (zwei über `ILOCK_T_FF_AX` gegenseitig verriegelte Toggle-Flip-Flops) um den bislang letzten noch ungenutzten Interlock-Baustein: `ILOCK_T_FF_SR_AX`. Dieser bietet zusätzlich zum Toggle-Eingang (`CLK`) zwei direkte Ereigniseingänge `S` (Set) und `R` (Reset) – nur an der ersten Instanz gezeigt. Von beiden wirkt nur `S` über die Verriegelungskette auf das jeweils andere Flip-Flop (genau wie ein CLK-Toggle nach EIN); `R` betrifft ausschließlich den lokalen Ausgang und lässt das Partner-Flip-Flop unverändert.

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_CLK_I1**: logiBUS Ereigniseingang (Typ: `logiBUS::io::DI::logiBUS_IE`)
  - **Parameter**: QI = TRUE, Input = Input_I1, InputEvent = BUTTON_SINGLE_CLICK
  - **Erklärung**: Klick auf I1 togglet FF1 (`ILOCK_T_FF_SR_1.CLK`).
- **DigitalInput_Set_I3**: logiBUS Ereigniseingang (Typ: `logiBUS::io::DI::logiBUS_IE`)
  - **Parameter**: QI = TRUE, Input = Input_I3, InputEvent = BUTTON_SINGLE_CLICK
  - **Erklärung**: Klick auf I3 setzt FF1 direkt (`ILOCK_T_FF_SR_1.S`), unabhängig vom vorherigen Zustand.
- **DigitalInput_Reset_I4**: logiBUS Ereigniseingang (Typ: `logiBUS::io::DI::logiBUS_IE`)
  - **Parameter**: QI = TRUE, Input = Input_I4, InputEvent = BUTTON_SINGLE_CLICK
  - **Erklärung**: Klick auf I4 setzt FF1 direkt zurück (`ILOCK_T_FF_SR_1.R`).
- **DigitalInput_CLK_I2**: logiBUS Ereigniseingang (Typ: `logiBUS::io::DI::logiBUS_IE`)
  - **Parameter**: QI = TRUE, Input = Input_I2, InputEvent = BUTTON_SINGLE_CLICK

  - **Erklärung**: Klick auf I2 togglet FF2 (`ILOCK_T_FF_SR_2.CLK`). Für FF2 sind keine direkten Set-/Reset-Eingänge verdrahtet.
- **ILOCK_T_FF_SR_1**, **ILOCK_T_FF_SR_2**: Verriegelbares Toggle-Flip-Flop mit Set/Reset (Typ: `logiBUS::signalprocessing::interlock::ILOCK_T_FF_SR_AX`)
  - **Parameter**: keine
  - **Erklärung**: Jede Instanz toggelt ihren Ausgang `Q` bei `CLK`, kann ihn aber auch direkt über `S`/`R` setzen bzw. zurücksetzen. `ILOCK_IN`/`ILOCK_OUT` sind vom **bidirektionalen** Adaptertyp `AE2` (ein Ereignis in jede Richtung, `EI1`/`EO1`) – die eine `Connection` `ILOCK_OUT`⟶`ILOCK_IN` reicht deshalb für gegenseitige Verriegelung: Wird eine Instanz eingeschaltet (per CLK-Toggle ODER per direktem `S`), wird die andere automatisch ausgeschaltet, unabhängig davon, welche der beiden Instanzen zuerst eingeschaltet wurde. Eine zusätzliche Rückverbindung wäre bei einem unidirektionalen Adapter (`AX`) nötig, bei `AE2` aber überflüssig (und würde beim Kompilieren als doppelte Verbindung kollidieren).
- **DigitalOutput_Q1**, **DigitalOutput_Q2**: logiBUS Digitalausgänge (Typ: `logiBUS::io::DQ::logiBUS_QXA`)
  - **Parameter**: QI = TRUE, Output = Output_Q1 bzw. Output_Q2
  - **Erklärung**: Geben die Zustände von `ILOCK_T_FF_SR_1.Q` bzw. `ILOCK_T_FF_SR_2.Q` an die Peripherie weiter.

Sub-Bausteine: keine

Die Übung verwendet keine weiteren Unterbausteine, alle FBs sind direkt auf der obersten Ebene der SubApp angeordnet.

Programmablauf und Verbindungen

1. `DigitalInput_CLK_I1.IND` ⟶ `ILOCK_T_FF_SR_1.CLK`: Klick auf I1 togglet FF1.
2. `DigitalInput_Set_I3.IND` ⟶ `ILOCK_T_FF_SR_1.S`: Klick auf I3 setzt FF1 direkt (ohne Toggle-Logik).
3. `DigitalInput_Reset_I4.IND` ⟶ `ILOCK_T_FF_SR_1.R`: Klick auf I4 setzt FF1 direkt zurück.
4. `DigitalInput_CLK_I2.IND` ⟶ `ILOCK_T_FF_SR_2.CLK`: Klick auf I2 togglet FF2.
5. `ILOCK_T_FF_SR_1.ILOCK_OUT` ⟶ `ILOCK_T_FF_SR_2.ILOCK_IN`: Die Verriegelungskette überträgt jeden Wechsel von FF1 auf EIN (per CLK-Toggle oder per direktem Set) an FF2 und schaltet es dadurch aus; ein Wechsel auf AUS (per CLK-Toggle oder per direktem Reset) wird dagegen nicht übertragen.
6. `ILOCK_T_FF_SR_1.Q` ⟶ `DigitalOutput_Q1.OUT`, `ILOCK_T_FF_SR_2.Q` ⟶ `DigitalOutput_Q2.OUT`.
7. **Testablauf**: I1 (CLK FF1) klicken ⟶ Q1 EIN, Q2 automatisch AUS (wie in `Uebung_206_AX`). I3 (Set FF1) klicken ⟶ dasselbe Ergebnis, aber ohne Toggle-Verhalten – Q1 ist danach garantiert EIN, unabhängig vom vorherigen Zustand, und verriegelt FF2 genauso wie ein CLK-Toggle. I4 (Reset FF1) klicken ⟶ Q1 AUS, **Q2 bleibt dabei unverändert**, da ein Reset kein Verriegelungssignal an die Kette sendet – anders als Toggle-auf-EIN oder direktes Set.

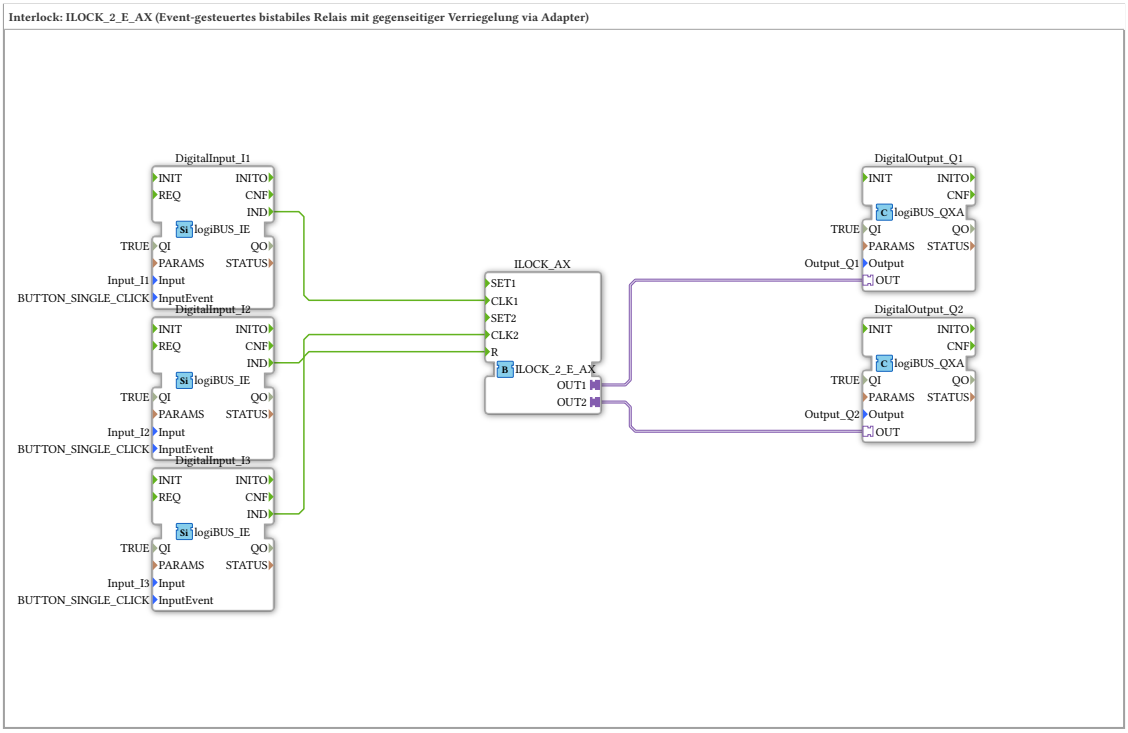
Zusammenfassung

Die Übung 206b_AX zeigt den letzten bislang ungenutzten Interlock-Baustein `ILOCK_T_FF_SR_AX`: Er erweitert das aus `Uebung_206_AX` bekannte gegenseitig verriegelte Toggle-Flip-Flop-Paar um direkte Set-/Reset-Eingänge. Wichtig ist die Asymmetrie im Verriegelungsverhalten: Ein Zustandswechsel auf EIN (per Toggle oder per direktem Set) wirkt stets verriegelnd auf das jeweils andere Flip-Flop, während ein direktes Reset die Verriegelungskette nicht beeinflusst und das Partner-Flip-Flop unverändert lässt.

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

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

2.1.2.485 Uebung_207_AX: Interlock: ILOCK_2_E_AX (Event-gesteuertes bistabiles Relais mit gegenseitiger Verriegelung via Adapter)



• — ○ • — ○ • — ○ •

Einleitung

Diese Übung demonstriert den Einsatz eines event-gesteuerten bistabilen Relais mit gegenseitiger Verriegelung (Interlock). Mithilfe des Funktionsbausteins `ILOCK_2_E_AX` wird ein einfaches Zwei-Kanal-Set/Reset-System aufgebaut, das über zwei digitale Eingänge getaktet wird und zwei digitale Ausgänge ansteuert. Ein dritter digitaler Eingang dient als separater Rücksetzeingang.

Verwendete Funktionsbausteine (FBs)

FB-Name	Typ	Beschreibung
`DigitalInput_I1`	`logiBUS::io::DI::logiBUS_IE`	Digitaler Eingang, parametriert mit `Input_I1` und Ereignis `BUTTON_SINGLE_CLICK`
`DigitalInput_I2`	`logiBUS::io::DI::logiBUS_IE`	Digitaler Eingang, parametriert mit `Input_I2` und Ereignis `BUTTON_SINGLE_CLICK`
`DigitalInput_I3`	`logiBUS::io::DI::logiBUS_IE`	Digitaler Eingang, parametriert mit `Input_I3` und Ereignis `BUTTON_SINGLE_CLICK`
`ILOCK_AX`	`logiBUS::signalprocessing::interlock::ILOCK_2_E_AX`	Interlock-Baustein mit zwei ereignisgesteuerten Set/Reset-Eingängen und Adapter-Ausgängen
`DigitalOutput_Q1`	`logiBUS::io::DQ::logiBUS_QXA`	Digitaler Ausgang, parametriert mit `Output_Q1`
`DigitalOutput_Q2`	`logiBUS::io::DQ::logiBUS_QXA`	Digitaler Ausgang, parametriert mit `Output_Q2`

Sub-Bausteine (keine eigenen Sub-Applikationen)

In dieser Übung werden keine benutzerdefinierten Sub-Bausteine verwendet – alle FBs stammen aus den Bibliotheken `logiBUS` (Digitale Ein-/Ausgabe und Signalverarbeitung).

Programmablauf und Verbindungen

1. **Ereignisverknüpfung**
   - Die Ereignisausgänge `IND` der drei `logiBUS_IE`-Eingänge sind wie folgt mit dem `ILOCK_2_E_AX` verbunden:
     - `DigitalInput_I1.IND` ⟶ `ILOCK_AX.CLK1` (Setzen von Kanal 1)
     - `DigitalInput_I2.IND` ⟶ `ILOCK_AX.CLK2` (Setzen von Kanal 2)
     - `DigitalInput_I3.IND` ⟶ `ILOCK_AX.R` (gemeinsamer Reset)
2. **Adapterverbindungen**
   - Die Ausgänge des Interlock-Bausteins werden über Adapterverbindungen an die digitalen Ausgänge weitergegeben:
     - `ILOCK_AX.OUT1` ⟶ `DigitalOutput_Q1.OUT`
     - `ILOCK_AX.OUT2` ⟶ `DigitalOutput_Q2.OUT`
3. **Funktionsweise**
   - Ein Ereignis (Single-Click) auf `I1` setzt den Ausgang `Q1` und löscht gleichzeitig `Q2` (gegenseitige Verriegelung).
   - Ein Ereignis auf `I2` setzt `Q2` und löscht `Q1`.
   - Ein Ereignis auf `I3` setzt beide Ausgänge zurück (`R = Reset`).
   - Der Interlock-Baustein arbeitet flankengesteuert: Nur bei eintreffenden Ereignissen ändern sich die Ausgangszustände.

Lernziele:

- Verständnis des Prinzips der gegenseitigen Verriegelung (Interlock) in der Automatisierungstechnik
- Umgang mit ereignisgesteuerten Funktionsbausteinen (event-gesteuert)
- Kommunikation über Adapterschnittstellen zwischen Signalverarbeitung und Ein-/Ausgabe
- Parametrierung von digitalen Eingängen mit verschiedenen Ereignistypen

Schwierigkeitsgrad: Mittel **Vorkenntnisse**: Grundlagen der IEC 61499, Bedienung der 4diac-IDE, Aufbau einfacher Netzwerke mit Ein-/Ausgabe-FBs

Ausführung: Nach dem Öffnen der Übung in der 4diac-IDE kann das System auf einer geeigneten Hardware (z.B. logiBUS-kompatibles System) gestartet werden. Die Taster an den Eingängen I1, I2 und I3 schalten die Ausgänge Q1 und Q2 entsprechend der Interlock-Logik.

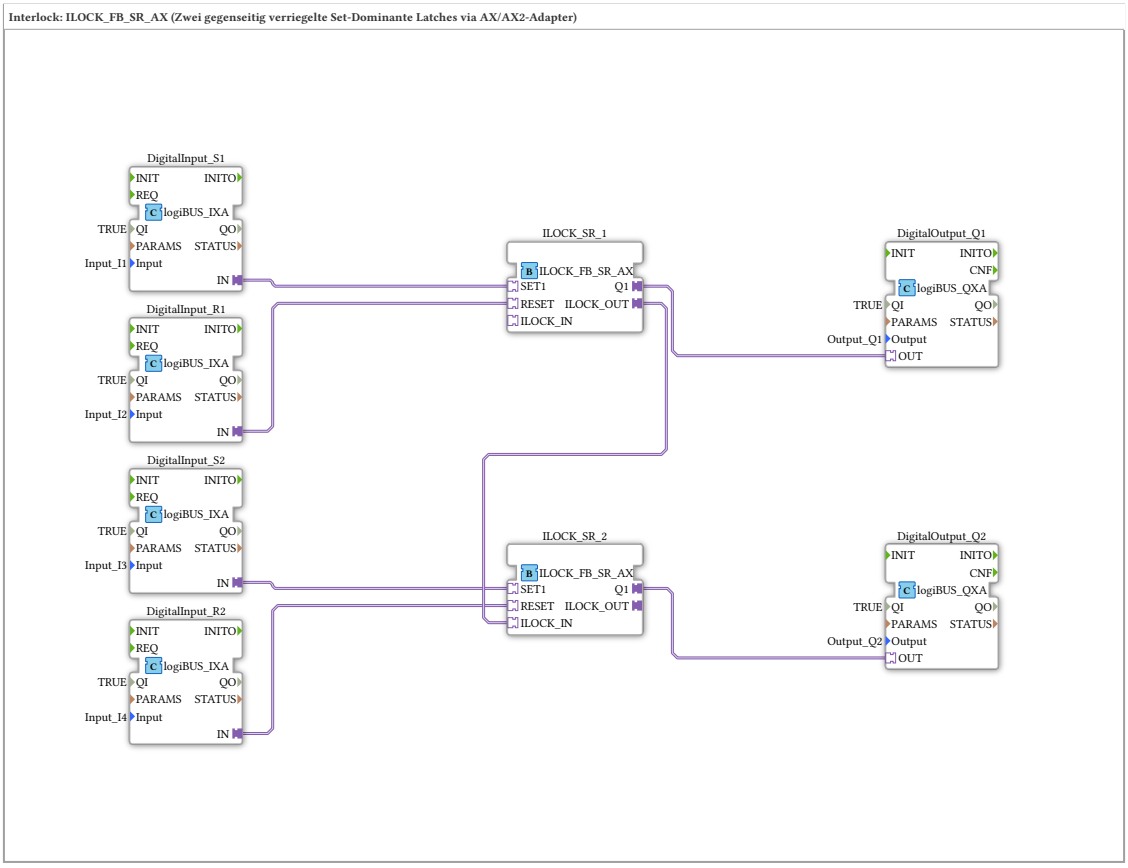
Zusammenfassung

Die Übung `Uebung_207_AX` vermittelt die praktische Anwendung eines event-gesteuerten Interlock-Bausteins. Sie zeigt, wie zwei Ausgänge wechselseitig gesperrt werden können und wie ein separater Reset-Eingang das System in einen definierten Grundzustand versetzt. Durch die Verwendung von Adapterverbindungen wird die lose Kopplung zwischen Signalverarbeitung und Peripherie deutlich – ein Kernkonzept der IEC 61499.

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

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

2.1.2.486 Uebung_208_AX: Interlock: ILOCK_FB_SR_AX (Zwei gegenseitig verriegelte Set-Dominante Latches via AX/AX2-Adapter)



- **Titel**: Uebung_208_AX *Interlock: ILOCK_FB_SR_AX (Zwei gegenseitig verriegelte Set-Dominante Latches via AX/AX2-Adapter)*

• – ○ • – ○ • – ○ •

Einleitung

Diese Übung veranschaulicht die Realisierung einer gegenseitigen Verriegelung (Interlock) zwischen zwei Set-dominanten SR-Latches. Die beiden Latches sind so miteinander verschaltet, dass immer nur einer der beiden Ausgänge aktiv sein kann. Die Ansteuerung erfolgt über digitale Eingänge, die Ausgänge werden über digitale Ausgänge ausgegeben. Die Verriegelung wird über die speziellen Adapter `ILOCK_IN` und `ILOCK_OUT` der Bausteine `ILOCK_FB_SR_AX` realisiert.

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: DigitalInput_S1, DigitalInput_R1, DigitalInput_S2, DigitalInput_R2

- **Typ**: `logiBUS::io::DI::logiBUS_IXA`
- **Verwendete interne FBs**: Keine (Hardware-Adapterbaustein)
- **Parameter**:
  - `QI` = `TRUE`
  - `Input` = `Input_I1` (bzw. `Input_I2`, `Input_I3`, `Input_I4`)
- **Funktionsweise**: Diese Bausteine lesen die digitalen Eingangssignale der logiBUS-Hardware ein und stellen sie über den Adapterausgang `IN` zur Verfügung. Sie dienen als Schnittstelle zu den physikalischen Eingängen.

Sub-Bausteine: ILOCK_SR_1, ILOCK_SR_2

- **Typ**: `logiBUS::signalprocessing::interlock::ILOCK_FB_SR_AX`
- **Verwendete interne FBs**: Keine (vordefinierter Interlock-Baustein)

- **Parameter**: Keine (Standardkonfiguration)
- **Funktionsweise**: Diese Bausteine realisieren je ein set-dominantes SR-Latch (Set-Dominant). Sie besitzen zwei Adapterschnittstellen:
  - `SET1`: Set-Eingang (aktiv-high)
  - `RESET`: Reset-Eingang (aktiv-high)
  - `Q1`: Ausgang (aktiv-high)
  - `ILOCK_IN`: Eingang für die Verriegelung von einem anderen Latch
  - `ILOCK_OUT`: Ausgang zur Verriegelung eines anderen Latches

  Die Verriegelung bewirkt, dass ein aktiver `ILOCK_IN` das Setzen des eigenen Latches blockiert. Dadurch kann immer nur eines der beiden Latches gesetzt sein.

Sub-Bausteine: DigitalOutput_Q1, DigitalOutput_Q2

- **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
- **Verwendete interne FBs**: Keine (Hardware-Adapterbaustein)
- **Parameter**:
  - `QI` = `TRUE`
  - `Output` = `Output_Q1` (bzw. `Output_Q2`)
- **Funktionsweise**: Diese Bausteine geben den Zustand des Eingangs `OUT` als digitales Ausgangssignal auf die logiBUS-Hardware aus. Sie dienen als Schnittstelle zu den physikalischen Ausgängen.

Programmablauf und Verbindungen

Die Übung besteht aus zwei identischen parallelen Zweigen, die über eine gegenseitige Verriegelung verbunden sind.

- **Erster Zweig**: Digitaleingang `DigitalInput_S1` (verbunden mit Hardware-Eingang `Input_I1`) liefert das Set-Signal an `ILOCK_SR_1.SET1`. Digitaleingang `DigitalInput_R1` (verbunden mit `Input_I2`) liefert das Reset-Signal an `ILOCK_SR_1.RESET`. Der Ausgang `ILOCK_SR_1.Q1` wird auf `DigitalOutput_Q1.OUT` gegeben und schaltet den Hardware-Ausgang `Output_Q1`.
- **Zweiter Zweig**: Analog wird `ILOCK_SR_2` durch `DigitalInput_S2` (über `Input_I3`) gesetzt und durch `DigitalInput_R2` (über `Input_I4`) zurückgesetzt. Sein Ausgang `Q1` steuert `Output_Q2`.
- **Interlock-Verbindung**: Der Ausgang `ILOCK_SR_1.ILOCK_OUT` ist mit dem Eingang `ILOCK_SR_2.ILOCK_IN` verbunden. Dadurch wird erreicht, dass wenn `ILOCK_SR_1` gesetzt ist (Q1 = TRUE), das Setzen von `ILOCK_SR_2` blockiert wird. Umgekehrt (Symmetrie) – in dieser Konfiguration ist nur eine Richtung explizit verdrahtet, die andere wird implizit durch die interne Logik des Bausteins realisiert. Tatsächlich besitzt `ILOCK_FB_SR_AX` eine bidirektionale Verriegelung: beide Latches blockieren sich gegenseitig. Die zusätzliche Verbindung des `ILOCK_OUT` des einen zum `ILOCK_IN` des anderen stellt sicher, dass immer nur einer aktiv ist.

Ablauf:

1. Liegt an `S1` ein TRUE-Signal an (und `ILOCK_SR_1` ist nicht durch `ILOCK_SR_2` blockiert), so wird `Q1` gesetzt.
2. Liegt an `S2` ein TRUE-Signal an (und `ILOCK_SR_2` ist nicht durch `ILOCK_SR_1` blockiert), so wird `Q2` gesetzt.
3. Ein Reset über `R1` bzw. `R2` setzt das jeweilige Latch zurück.
4. Durch die Verriegelung kann nie gleichzeitig `Q1` und `Q2` TRUE sein. Ein Setzversuch am blockierten Latch bleibt wirkungslos.

Lernziele:

- Verständnis von gegenseitigen Verriegelungen (Interlocks) in der Steuerungstechnik.
- Anwendung set-dominanter SR-Latches.
- Umgang mit Adapterverbindungen in 4diac-IDE (AX/AX2-Konzept).

Schwierigkeitsgrad: Mittel **Vorkenntnisse**: Grundlagen der logiBUS-Ein-/Ausgabe, SR-Latch-Funktionalität.

Zusammenfassung

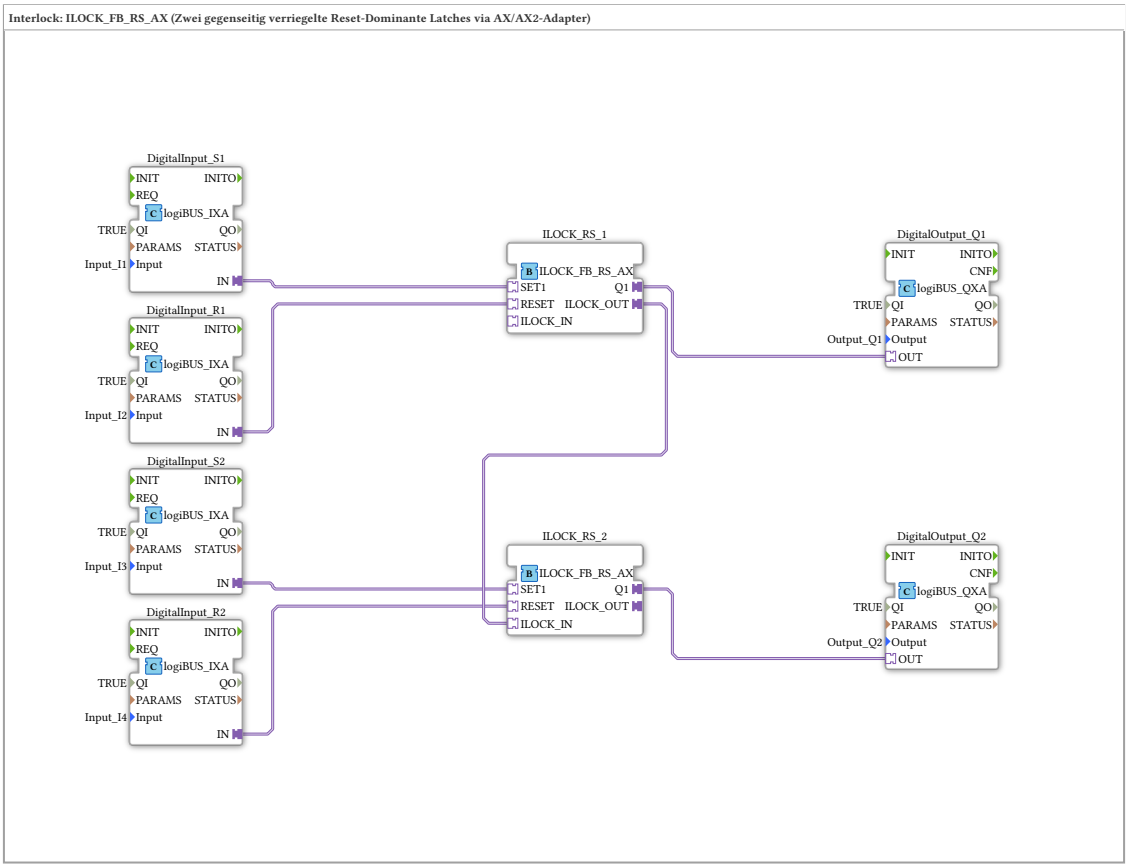
Die Übung `Uebung_208_AX` demonstriert den Aufbau einer gegenseitigen Verriegelung zwischen zwei set-dominanten Latches mithilfe der speziellen Interlock-Bausteine `ILOCK_FB_SR_AX`. Durch die Adapter-

verbindungen `ILOCK_IN` und `ILOCK_OUT` wird sichergestellt, dass stets nur einer der beiden Ausgänge aktiv sein kann. Dies ist typisch für Anwendungen, bei denen sich zwei Zustände gegenseitig ausschließen müssen (z.B. Motorenrichtungen, Ventilsteuerungen). Die Übung festigt den Umgang mit Hardware-Adaptern und den Interlock-Mechanismen der logiBUS-Bibliothek.

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

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

2.1.2.487 Uebung_209_AX: Interlock: ILOCK_FB_RS_AX (Zwei gegenseitig verriegelte Reset-Dominante Latches via AX/AX2-Adapter)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert den Aufbau einer **gegenseitigen Verriegelung (Interlock)** zwischen zwei reset-dominanten RS-Latches. Die Schaltung verhindert, dass beide Ausgänge gleichzeitig aktiv werden – ein typisches Sicherheitsmerkmal in der Steuerungstechnik. Die Realisierung erfolgt über zwei **ILOCK_FB_RS_AX**-Blöcke, die über einen AX/AX2-Adapter miteinander verbunden sind. Jeweils ein Setz- und ein Rücksetzeingang steuern die Latches, während die Ausgänge die digitalen Ausgänge Q1 und Q2 ansteuern.

Verwendete Funktionsbausteine (FBs)

In der Übung werden folgende Funktionsbausteine eingesetzt:

FB-Name	Typ	Zweck
`DigitalInput_S1`	`logiBUS_IXA`	Einlesen des Setz-Signals S1 (Eingang I1)
`DigitalInput_R1`	`logiBUS_IXA`	Einlesen des Rücksetz-Signals R1 (I2)
`DigitalInput_S2`	`logiBUS_IXA`	Einlesen des Setz-Signals S2 (I3)

DigitalInput_R2	logiBUS_IXA	Einlesen des Rücksetz-Signals R2 (I4)
ILOCK_RS_1	ILOCK_FB_RS_AX	Erstes verriegeltes RS-Latch
ILOCK_RS_2	ILOCK_FB_RS_AX	Zweites verriegeltes RS-Latch
DigitalOutput_Q1	logiBUS_QXA	Ausgabe an Q1
DigitalOutput_Q2	logiBUS_QXA	Ausgabe an Q2

Sub-Bausteine: `ILOCK_FB_RS_AX`

- **Typ**: Bibliotheksbaustein aus der logiBUS-Bibliothek (`logiBUS::signalprocessing::interlock::ILOCK_FB_RS_AX`)
- **Verwendete interne FBs** (Konzeptuell):
  - Zwei reset-dominante RS-Latches (SR)
  - Ein Interlock-Gatter, das die Ausgänge der Latches gegenseitig sperrt
- **Schnittstellen (Adapter)**:
  - `SET1`: Setz-Eingang über AX-Adapter
  - `RESET`: Rücksetz-Eingang über AX-Adapter
  - `Q1`: Ausgang des Latches (AX-Adapter)
  - `ILOCK_IN`: Eingang für die Verriegelung vom anderen Latch
  - `ILOCK_OUT`: Ausgang, der den eigenen Zustand an den anderen Latch meldet
- **Funktionsweise**: Der Baustein realisiert ein reset-dominantes RS-Latch, dessen Ausgang `Q1` gesetzt wird, wenn `SET1` aktiv ist und kein aktives RESET anliegt. Der Ausgang bleibt gesetzt, bis `RESET` aktiv wird (Reset-Dominanz). Zusätzlich wird über `ILOCK_IN` der Zustand des anderen Latches empfangen: Wenn der andere Latch aktiv ist, wird das Setzen des eigenen Latches unterbunden. Der eigene Zustand wird über `ILOCK_OUT` an den anderen Latch weitergegeben.

Programmablauf und Verbindungen

Die Verdrahtung im `SubAppNetwork` erfolgt über Adapterverbindungen:

1. **Eingangsverarbeitung**:
   - `DigitalInput_S1` liefert den Setz-Befehl für `ILOCK_RS_1` (S1).
   - `DigitalInput_R1` liefert den Rücksetz-Befehl für `ILOCK_RS_1` (R1).
   - Analog für die zweite Gruppe: `DigitalInput_S2` → `ILOCK_RS_2.SET1`, `DigitalInput_R2` → `ILOCK_RS_2.RESET`.
2. **Interlock-Verkettung**:
   - Der Ausgang `ILOCK_RS_1.ILOCK_OUT` ist mit `ILOCK_RS_2.ILOCK_IN` verbunden.
   - Diese Verbindung stellt sicher, dass `ILOCK_RS_2` nur dann gesetzt werden kann, wenn `ILOCK_RS_1` nicht aktiv ist (bzw. umgekehrt, da der zweite Block ebenfalls seinen ILOCK_OUT angeben müsste – in dieser Konfiguration ist nur eine Richtung explizit verdrahtet, die interne Logik berücksichtigt jedoch die gegenseitige Sperre).
3. **Ausgangssteuerung**:
   - `ILOCK_RS_1.Q1` steuert über `DigitalOutput_Q1` den Ausgang Q1.
   - `ILOCK_RS_2.Q1` steuert über `DigitalOutput_Q2` den Ausgang Q2.

Ablauf: Ein Taster an S1 setzt den ersten Latch (Q1 ein), solange R1 nicht gedrückt ist. Wird S2 betätigt, kann der zweite Latch nur aktiv werden, wenn der erste Latch inaktiv ist (durch Interlock). Erst nach einem Reset des ersten Latches (R1) kann der zweite Latch gesetzt werden. Dadurch wird ein gleichzeitiges Einschalten beider Ausgänge ausgeschlossen.

Zusammenfassung

Die Übung **Uebung_209_AX** vermittelt das grundlegende Prinzip eines **Interlocks** mit zwei reset-dominanten RS-Latches. Durch die Verwendung des vorgefertigten Bausteins `ILOCK_FB_RS_AX` und der AX-Adapter wird die Verriegelung einfach und übersichtlich realisiert. Lernziele sind:

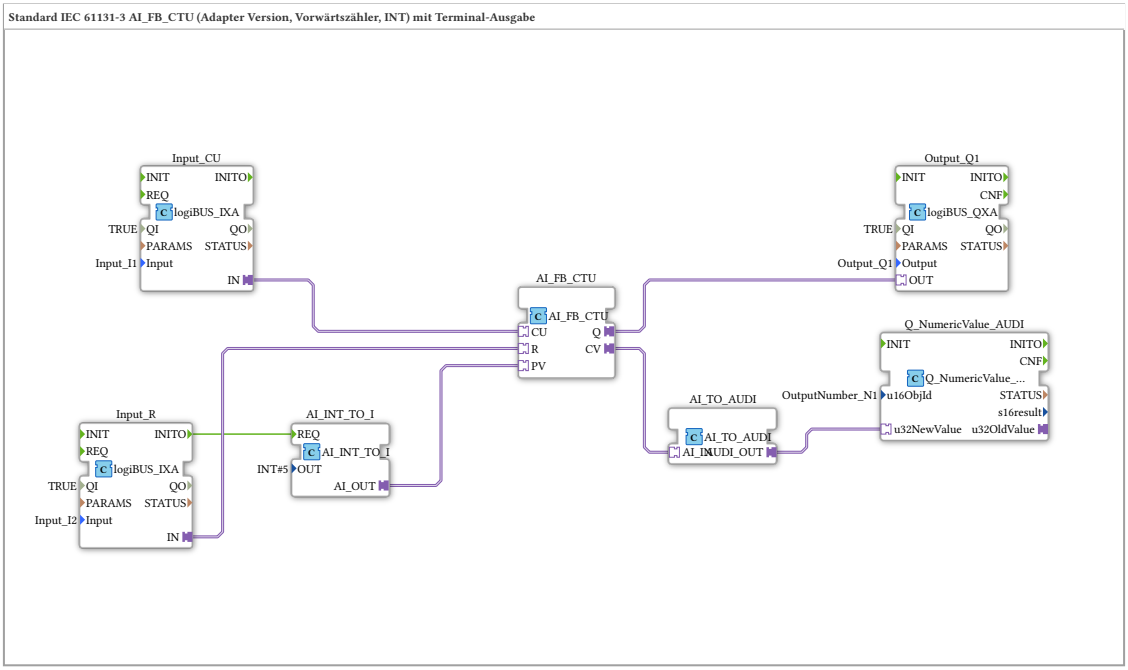
- Verständnis des Interlock-Mechanismus in der Steuerungstechnik
- Umgang mit Adapter-basierten Verbindungen in 4diac
- Erkennen von Sicherheitsanforderungen (gegenseitige Sperre)

Diese Schaltung findet beispielsweise Anwendung bei der Ansteuerung von zwei gegenläufigen Motoren oder bei Zustandsmaschinen, bei denen nur ein Zustand gleichzeitig aktiv sein darf.

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

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

2.1.2.488 Uebung_210_AI: Standard IEC 61131-3 AI_FB_CTU (Adapter Version, Vorwärtszähler, INT) mit Terminal-Ausgabe



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung implementiert einen Vorwärtszähler (CTU) gemäß IEC 61131-3 in einer Adapter-basierten Variante. Der gezählte Wert wird über ein Terminal ausgegeben. Die Schaltung zeigt die Interaktion zwischen logiBUS-Eingängen, einem Zählerbaustein, Konvertierungsbausteinen und einer Terminalausgabe.

Verwendete Funktionsbausteine (FBs)

FB: AI_FB_CTU

- **Typ**: `adapter::iec61131::counters::AI_FB_CTU`
- **Funktion**: Vorwärtszähler mit Adapterschnittstelle. Zählt bei jedem positiven Ereignis am Eingang CU den internen Zähler um 1 hoch. Der Zähler wird über den Eingang R zurückgesetzt. Der aktuelle Zählerwert (CV) und der Ausgang Q (wenn CV ≥ PV) werden über Adapterausgänge bereitgestellt.

FB: AI_INT_TO_I

- **Typ**: `adapter::conversion::unidirectional::AI_INT_TO_I`
- **Parameter**: `OUT = INT#5` (fester Preset-Wert)
- **Funktion**: Wandelt einen konstanten Integer-Wert (5) in das erforderliche Adapterformat um und stellt ihn als Preset-Wert (PV) für den Zähler bereit.

FB: Input_CU

- **Typ**: `logiBUS::io::DI::logiBUS_IXA`
- **Parameter**: `QI = TRUE`, `Input = Input_I1`
- **Funktion**: Liest den digitalen Eingang `Input_I1` und stellt ihn über einen Adapterausgang als Zählimpuls (CU) bereit.

FB: Input_R

- **Typ**: `logiBUS::io::DI::logiBUS_IXA`
- **Parameter**: `QI = TRUE`, `Input = Input_I2`
- **Funktion**: Liest den digitalen Eingang `Input_I2` und stellt ihn über einen Adapterausgang als Rücksetzsignal (R) bereit.

FB: Output_Q1

- **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
- **Parameter**: `QI = TRUE`, `Output = Output_Q1`
- **Funktion**: Steuert den digitalen Ausgang `Output_Q1`. Der Ausgang wird aktiv, sobald der Zähler den Preset-Wert erreicht oder überschreitet.

FB: AI_TO_AUDI

- **Typ**: `adapter::conversion::unidirectional::AI_TO_AUDI`
- **Funktion**: Wandelt den analogen Zählerwert (CV) in das AUDI-Format um, das für die Terminalausgabe benötigt wird. **Hinweis**: Der Baustein kann keine negativen Zahlen darstellen.

FB: Q_NumericValue_AUDI

- **Typ**: `isobus::UT::Q::Q_NumericValue_AUDI`
- **Parameter**: `u16ObjId = OutputNumber_N1`
- **Funktion**: Gibt den übergebenen numerischen Wert auf dem Terminal aus. Die Objekt-ID `OutputNumber_N1` definiert die Position der Anzeige.

Programmablauf und Verbindungen

Die Schaltung arbeitet wie folgt zusammen:

1. **Eingangssignale**:
   - Der digitale Eingang `Input_I1` wird über `Input_CU` als Zählimpuls (CU) an den Zähler `AI_FB_CTU` weitergeleitet.
   - Der digitale Eingang `Input_I2` wird über `Input_R` als Rücksetzsignal (R) an den Zähler weitergeleitet.
2. **Preset-Wert**:
   - Der Baustein `AI_INT_TO_I` liefert einen festen Wert von 5. Dieser wird einmalig über eine Ereignisverbindung von `Input_R.INIT0` (Initialisierungsereignis) zu `AI_INT_TO_I.REQ` gesetzt und dann als Preset-Wert (PV) an den Zähler übergeben.
3. **Zählerverhalten**:
   - Bei jeder steigenden Flanke an CU wird der interne Zähler um 1 erhöht.
   - Ein positiver Impuls an R setzt den Zähler auf 0 zurück.
   - Erreicht der Zählerstand (CV) den Preset-Wert (5), wird der Ausgang Q gesetzt.
   - Der Ausgang Q wird über den Adapterausgang an den digitalen Ausgang `Output_Q1` geleitet.
4. **Terminalausgabe**:
   - Der aktuelle Zählerstand (CV) wird über `AI_TO_AUDI` in das AUDI-Format konvertiert.
   - Anschließend wird der Wert über `Q_NumericValue_AUDI` auf dem Terminal dargestellt.
5. **Hinweise aus den Kommentaren**:
   - Der Baustein `AI_TO_AUDI` unterstützt keine negativen Zahlen (kann bei bestimmten Anwendungen zu Fehlern führen).
   - Um die Ereignisrate zu reduzieren, kann optional ein `AX_D_FF` (D-Flipflop) zwischengeschaltet werden.

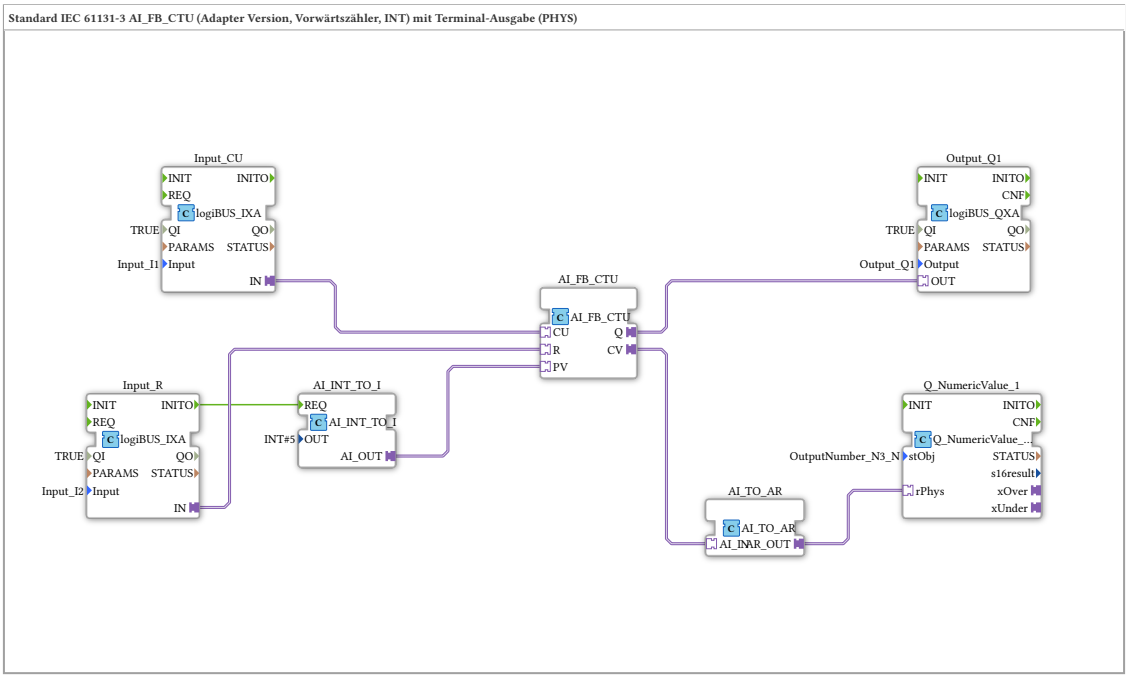
Zusammenfassung

Die Übung demonstriert die Realisierung eines IEC 61131-3 Vorwärtszählers mit Adapterschnittstelle (CTU) in 4diac. Die Ein- und Ausgänge sind über logiBUS-Komponenten an die Hardware angebunden. Der Zählerstand wird kontinuierlich auf einem Terminal ausgegeben, während der Ausgang Q einen digitalen Ausgang ansteuert. Die Konfiguration zeigt den Umgang mit Adapter-basierten Funktionsbausteinen, Typkonvertierungen und Terminalausgabe.

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

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

2.1.2.489 Uebung_210b_AR: Standard IEC 61131-3 AI_FB_CTU (Adapter Version, Vorwärtszähler, INT) mit Terminal-Ausgabe (PHYS)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung zeigt die Verwendung eines IEC 61131-3 Vorwärtszählers (CTU) in einer Adapter-Version. Der Zähler wird über zwei digitale Eingänge (CU für Zählimpulse, R für Reset) gesteuert. Der aktuelle Zählerstand wird über einen Analogausgang auf ein Terminal ausgegeben (PHYS). Ein Voreinstellwert (PV) wird beim Start auf 5 gesetzt. Der Ausgang Q des Zählers wird auf einen digitalen Ausgang geführt.

Verwendete Funktionsbausteine (FBs)

- **AI_FB_CTU** Typ: `adapter::iec61131::counters::AI_FB_CTU` Vorwärtszähler (CTU) mit INT-Datentyp. Zählt bei jedem positiven Flanke am CU-Eingang hoch und setzt den Ausgang Q, wenn CV >= PV.
- **AI_INT_TO_I** Typ: `adapter::conversion::unidirectional::AI_INT_TO_I` Wandelt einen konstanten Integer-Wert in ein IBN-konformes Signal. Parameter: `OUT = INT#5` (Voreinstellwert).
- **Input_CU** Typ: `logiBUS::io::DI::logiBUS_IXA` Digitaler Eingang für die Zählimpulse, verbunden mit `Input_I1`. Parameter: `QI = TRUE`.
- **Input_R** Typ: `logiBUS::io::DI::logiBUS_IXA` Digitaler Eingang für den Reset, verbunden mit `Input_I2`. Parameter: `QI = TRUE`.
- **Output_Q1** Typ: `logiBUS::io::DQ::logiBUS_QXA` Digitaler Ausgang, verbunden mit `Output_Q1`. Aktiv, wenn Zählerstand ≥ PV.
- **AI_TO_AR** Typ: `adapter::conversion::unidirectional::AI_TO_AR` Wandelt den analogen Zählerstand (CV) in einen AR-Wert für die Terminalausgabe.
- **Q_NumericValue_1** Typ: `isobus::UT::Q::Q_NumericValue_PHYSA` Terminal-Ausgabe (PHYS), zeigt den Zählerstand numerisch an. Parameter: `stObj = OutputNumber_N3`.

Programmablauf und Verbindungen

1. **Initialisierung** Beim Start wird über den Ereignisausgang `Input_R.INITO` der Baustein `AI_INT_TO_I` getriggert. Dieser liefert den konstanten Wert 5 an den PV-Eingang des Zählers.
2. **Zählen** Jeder positive Flanke am digitalen Eingang `Input_CU` (verbunden mit `Input_I1`) erhöht den Zählerstand (CV) um 1.
3. **Reset** Ein Signal am Eingang `Input_R` (verbunden mit `Input_I2`) setzt den Zählerstand auf 0 zurück.
4. **Ausgang Q** Wenn CV >= PV (5), wird der Ausgang Q aktiv. Dieser ist mit dem digitalen Ausgang `Output_Q1` verbunden.
5. **Anzeige am Terminal** Der aktuelle Zählerstand (CV) wird über `AI_TO_AR` in einen AR-Wert gewandelt und an die Terminalausgabe `Q_NumericValue_1` gesendet. Dadurch kann der Wert auf einem physischen Display oder einer Visualisierung angezeigt werden.

Hinweise aus der Konfiguration:

- Im Kommentar wird darauf hingewiesen, dass negative Werte möglich sind.
- Für eine Reduzierung der Ereignisrate könnte ggf. ein `AX_D_FF`-Baustein eingefügt werden.

Zusammenfassung

Die Übung vermittelt den Umgang mit einem IEC 61131-3-Zähler (CTU) in einer adapterbasierten Umgebung. **Lernziele:**

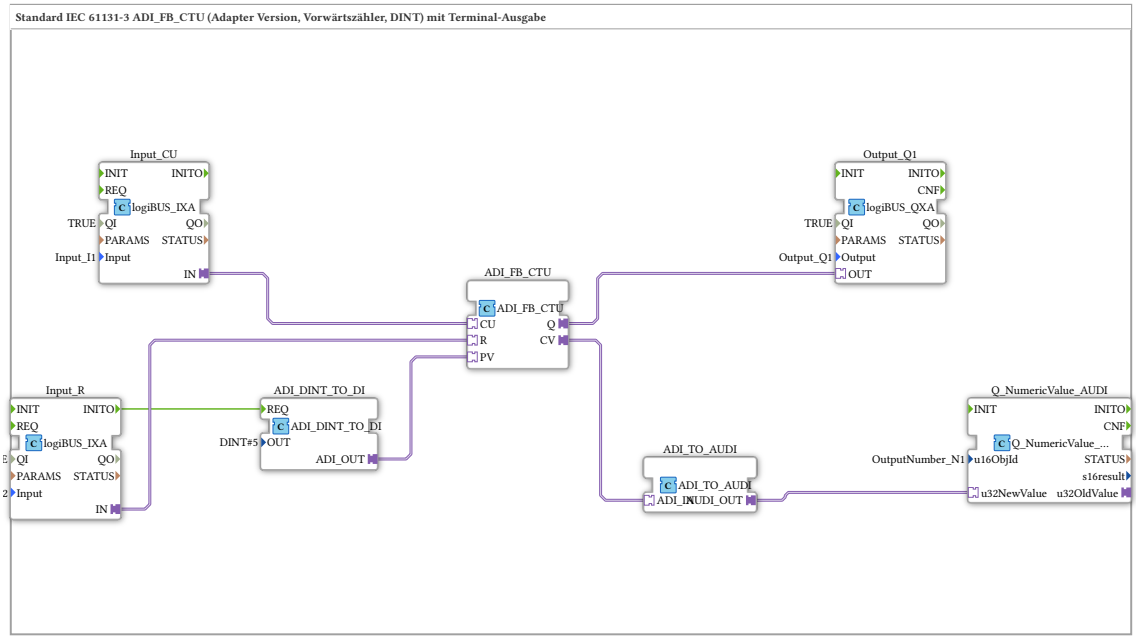
- Aufbau eines Vorwärtszählers mit Preset und Reset.
- Initialisierung eines Voreinstellwerts über einen Konstanten-Baustein.
- Anbindung digitaler Ein- und Ausgänge über logiBUS.
- Ausgabe eines Zählerstands auf ein Terminal (PHYS).

Schwierigkeitsgrad: Mittel **Vorkenntnisse:** Grundlagen der 4diac-IDE, Umgang mit logiBUS-Ein-/Ausgängen.

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

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

2.1.2.490 Uebung_211_ADI: Standard IEC 61131-3 ADI_FB_CTU (Adapter Version, Vorwärtszähler, DINT) mit Terminal-Ausgabe



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung implementiert einen Standard IEC 61131-3 Vorwärtszähler (Counter Up, CTU) als Adapter-Version für den Datentyp DINT. Der Zählerstand wird zusätzlich über ein Terminal ausgegeben. Die Hardware-Eingänge (CU und R) werden über logiBUS DI-Bausteine eingelesen, der Ausgang Q steuert eine logiBUS DO-Klemme. Der Zählerendwert (PV) wird durch einen konstanten Wert von 5 festgelegt.

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: ADI_FB_CTU

- **Typ**: adapter::iec61131::counters::ADI_FB_CTU
- **Verwendete interne FBs**: Keine (Basisfunktionsbaustein)
- **Parameter**: Keine
- **Ereigniseingänge**: CU (Zählimpuls), R (Reset)
- **Datenausgänge**: Q (Ausgang, wenn CV >= PV), CV (aktueller Zählerstand)
- **Dateneingänge**: PV (Endwert)
- **Funktionsweise**: Der Baustein zählt bei jedem positiven Flanke am Ereigniseingang CU den internen Zähler CV hoch (DINT). Erreicht CV den Wert PV, wird Q gesetzt. Ein Signal am Eingang R setzt CV zurück auf 0 und Q zurück.

Sub-Bausteine: ADI_DINT_TO_DI

- **Typ**: adapter::conversion::unidirectional::ADI_DINT_TO_DI
- **Verwendete interne FBs**: Keine
- **Parameter**: `OUT = DINT#5` (fester Endwert)
- **Funktionsweise**: Wandelt einen DINT-Wert in einen Adapter-Datenausgang (ADI) um. Hier wird der konstante Wert 5 als PV für den Zähler bereitgestellt.

Sub-Bausteine: ADI_TO_AUDI

- **Typ**: adapter::conversion::unidirectional::ADI_TO_AUDI
- **Verwendete interne FBs**: Keine
- **Parameter**: Keine
- **Funktionsweise**: Konvertiert den Adapter-Datenausgang (ADI) in einen AUDI-Datenausgang, der für die Terminalausgabe geeignet ist. (Hinweis: Der Baustein unterstützt keine negativen Zahlen – siehe Kommentar im Netzwerk.)

Sub-Bausteine: Q_NumericValue_AUDI

- **Typ**: isobus::UT::Q::Q_NumericValue_AUDI
- **Verwendete interne FBs**: Keine
- **Parameter**: `u16ObjId = OutputNumber_N1` (Referenz auf das Terminal-Ausgabeobjekt)
- **Funktionsweise**: Nimmt einen AUDI-Datenwert (u32NewValue) entgegen und gibt diesen über das Terminal aus. Die Objekt-ID verweist auf den vordefinierten Ausgabeplatz.

Sub-Bausteine: Input_CU

- **Typ**: logiBUS::io::DI::logiBUS_IXA
- **Verwendete interne FBs**: Keine
- **Parameter**: `QI = TRUE`, `Input = Input_I1` (physischer Eingang)
- **Funktionsweise**: Liest den digitalen Eingang I1 und stellt ihn als Adapter-Datenausgang (IN) für den Zählimpuls CU bereit. Der Baustein ist immer aktiviert (QI = TRUE).

Sub-Bausteine: Input_R

- **Typ**: logiBUS::io::DI::logiBUS_IXA
- **Verwendete interne FBs**: Keine
- **Parameter**: `QI = TRUE`, `Input = Input_I2` (physischer Eingang)
- **Funktionsweise**: Liest den digitalen Eingang I2 und stellt ihn als Adapter-Datenausgang (IN) für den Reset R bereit. Zusätzlich löst der Ereignisausgang INITO eine einmalige Initialisierung des PV-Werts aus.

Sub-Bausteine: Output_Q1

- **Typ**: logiBUS::io::DQ::logiBUS_QXA
- **Verwendete interne FBs**: Keine

- **Parameter**: `QI = TRUE`, `Output = Output_Q1` (physischer Ausgang)
- **Funktionsweise**: Nimmt den Adapter-Dateneingang (OUT) vom Zählerausgang Q entgegen und setzt den physischen Ausgang Q1 entsprechend.

Programmablauf und Verbindungen

1. **Initialisierung**: Beim Start wird vom Baustein Input_R das Ereignis INITO ausgelöst. Dieses triggert den Baustein ADI_DINT_TO_DI, der den festen PV-Wert (5) als DINT bereitstellt und über den Adapterausgang ADI_OUT an den PV-Eingang von ADI_FB_CTU sendet.
2. **Zählimpulse**: Ein positiver Flanke am Eingang I1 (logiBUS) wird über Input_CU als Adaptersignal (IN) an den CU-Eingang des Zählers weitergeleitet. Der Zähler inkrementiert bei jedem Ereignis seinen internen CV.
3. **Reset**: Ein positiver Flanke am Eingang I2 wird über Input_R als Adaptersignal an den R-Eingang des Zählers weitergeleitet und setzt CV zurück.
4. **Ausgang**: Der Zählerausgang Q (gesetzt wenn CV >= PV) wird über eine Adapterverbindung an den OUT-Eingang von Output_Q1 weitergegeben und schaltet den physischen Ausgang Q1.
5. **Terminalausgabe**: Der aktuelle Zählerstand CV wird über ADI_TO_AUDI in ein AUDI-Format gewandelt und an Q_NumericValue_AUDI übergeben. Der Baustein gibt den Wert auf dem Terminal aus (Objekt OutputNumber_N1).

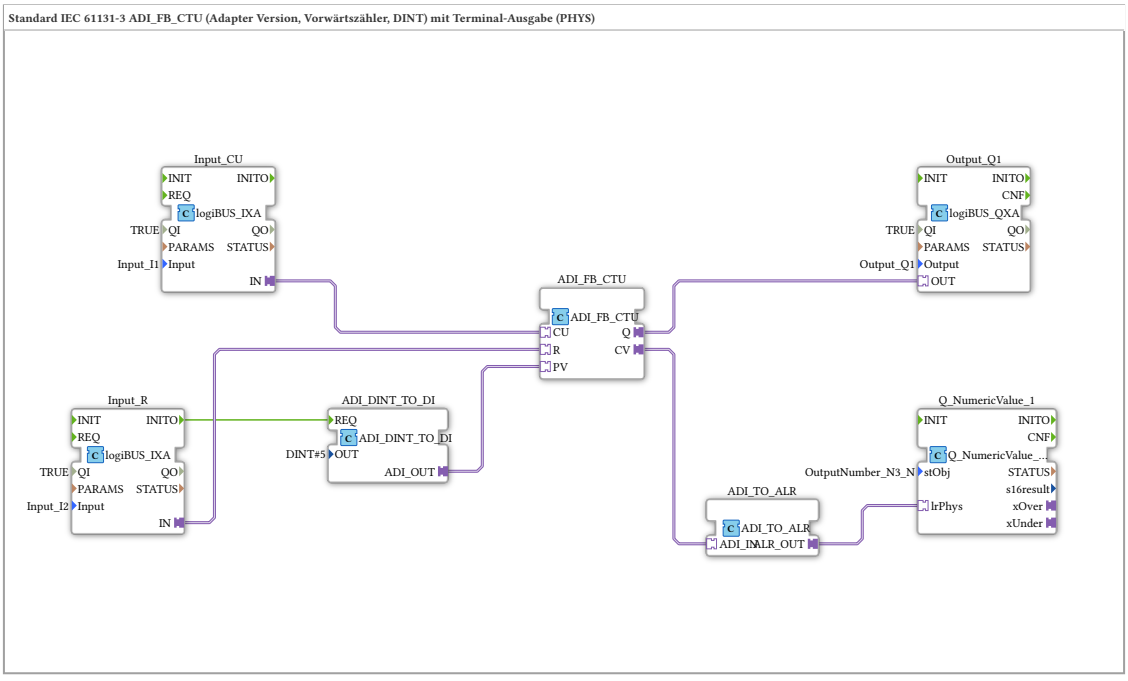
Hinweise:

- Der Kommentar im Netzwerk weist darauf hin, dass ADI_TO_AUDI keine negativen Zahlen verarbeiten kann – hier jedoch nicht relevant, da CV ≥ 0 ist.
- Ein möglicher AX_D_FF könnte die Ereignisrate reduzieren, falls die Zählimpulse zu schnell kommen.

Zusammenfassung

Die Übung demonstriert den Einsatz eines adaptierbaren Vorwärtszählers (ADI_FB_CTU) nach IEC 61131-3. Durch die Verwendung von Adapter-Bausteinen wird eine flexible und standardisierte Schnittstelle geschaffen. Der Zählerstand wird sowohl als Binärausgang (Q1) als auch numerisch auf einem Terminal ausgegeben, was die Fehlersuche und Visualisierung erleichtert. Die Konfiguration der Ein-/Ausgänge erfolgt über logiBUS-Klemmen und ist für den realen Einsatz vorbereitet.

2.1.2.491 Uebung_211b_ALR: Standard IEC 61131-3 ADI_FB_CTU (Adapter Version, Vorwärtszähler, DINT) mit Terminal-Ausgabe (PHYS)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung realisiert einen Vorwärtszähler nach IEC 61131-3 (ADI_FB_CTU) im Adapter-Format. Der Zählerwert wird auf einem Terminal (PHYS) ausgegeben. Die Konfiguration ermöglicht das Zählen von Impulsen, Rücksetzen des Zählers und die Darstellung des aktuellen Zählwertes inklusive negativer Werte.

Verwendete Funktionsbausteine (FBs)

- **ADI_FB_CTU** Typ: `adapter::iec61131::counters::ADI_FB_CTU` Vorwärtszähler-Baustein (DINT). Zählt bei jedem positiven Flanke am Eingang CU. Der aktuelle Zählerstand wird am Ausgang CV ausgegeben. Bei Erreichen des an PV angelegten Sollwerts wird der Ausgang Q gesetzt. Der Eingang R setzt den Zähler zurück.
- **ADI_DINT_TO_DI** Typ: `adapter::conversion::unidirectional::ADI_DINT_TO_DI` Wandelt einen DINT-Wert in einen DI-Datenstrom um. Parameter: OUT = DINT#5 (initialer Sollwert für den Zähler).
- **Input_CU** Typ: `logiBUS::io::DI::logiBUS_IXA` Digitaler Eingang für den Zählimpuls (CU). Parameter: QI = TRUE, Input = Input_I1.
- **Input_R** Typ: `logiBUS::io::DI::logiBUS_IXA` Digitaler Eingang für den Rücksetzbefehl (R). Parameter: QI = TRUE, Input = Input_I2.
- **Output_Q1** Typ: `logiBUS::io::DQ::logiBUS_QXA` Digitaler Ausgang für das Zählererreignis (Q). Parameter: QI = TRUE, Output = Output_Q1.
- **ADI_TO_ALR** Typ: `adapter::conversion::unidirectional::ADI_TO_ALR` Wandelt den ADI-Datenstrom des Zählerstandes in ein ALR-Format um (für die Terminalausgabe).
- **Q_NumericValue_1** Typ: `isobus::UT::Q::Q_NumericValue_PHYSA_LREAL` Gibt den Zählerstand als numerischen Wert auf dem Terminal (PHYS) aus. Parameter: stObj = OutputNumber_N3.

Programmablauf und Verbindungen

Der Programmablauf ist ereignisgesteuert:

1. **Initialisierung**: Beim Start wird der Sollwert (PV) durch den Baustein ADI_DINT_TO_DI auf DINT#5 gesetzt. Dies erfolgt über die Ereignisverbindung von `Input_R.INITO` zu `ADI_DINT_TO_DI.REQ`.
2. **Zählimpulse**: Der digitale Eingang Input_CU (I1) liefert Zählimpulse über den Adapter `Input_CU.IN` an den Zählereingang `ADI_FB_CTU.CU`.
3. **Rücksetzen**: Der digitale Eingang Input_R (I2) liefert Rücksetzsignale über `Input_R.IN` an den R-Eingang `ADI_FB_CTU.R`.
4. **Zählerausgang**: Der Ausgang Q des Zählers wird über `ADI_FB_CTU.Q` an den digitalen Ausgang `Output_Q1.OUT` weitergegeben (z.B. für eine Anzeige oder Steuerung).
5. **Zählerstand**: Der aktuelle Zählerwert (CV) wird über `ADI_FB_CTU.CV` an `ADI_TO_ALR.ADI_IN` übergeben. Der Baustein ADI_TO_ALR wandelt das Format in ALR um und leitet es an `Q_NumericValue_1.lrPhys` weiter. Dies ermöglicht die Ausgabe des Zählerstandes auf dem Terminal (PHYS).

Hinweise aus der Implementierung:

- Negative Zählerwerte sind möglich.
- Bei hohen Ereignisraten kann ein AX_D_FF (Event-Filter) zwischengeschaltet werden, um die Eventlast zu reduzieren.

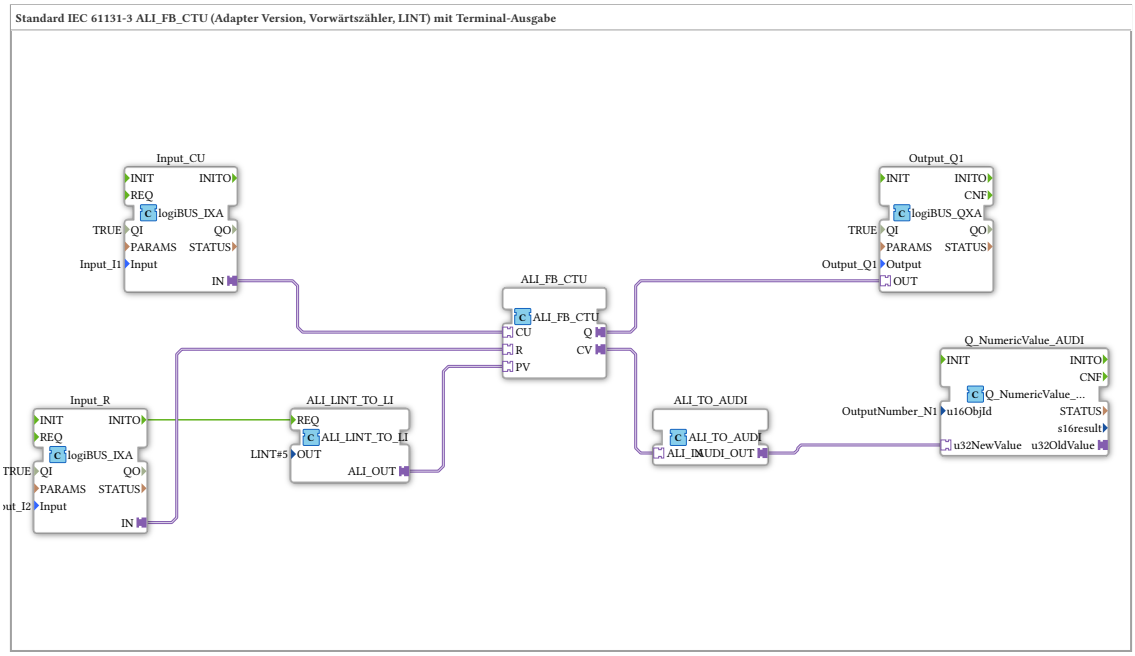
Zusammenfassung

Die Übung demonstriert die Verwendung eines standardisierten IEC 61131-3 Vorwärtszählers (ADI_FB_CTU) in der 4diac-IDE mit Adaptertechnologie. Die Verbindung von digitalen Ein-/Ausgängen, Konvertierungsbausteinen und Terminalausgabe zeigt eine typische Industriesteuerungsaufgabe. Lernziele sind das Verständnis von Zählerlogik, Ereignissteuerung und der Integration physischer I/O in ein Funktionsbaustein-Netzwerk. Voraussetzungen sind Grundkenntnisse der 4diac-IDE und der IEC 61131-3 Adapter-Bausteine.

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

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

2.1.2.492 Uebung_212_ALI: Standard IEC 61131-3 ALI_FB_CTU (Adapter Version, Vorwärtszähler, LINT) mit Terminal-Ausgabe



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert die Verwendung eines Vorwärtszählers (CTU) nach IEC 61131-3 in der Adapter-Version (ALI_FB_CTU). Der Zähler zählt Eingangsimpulse (CU) aufwärts, kann über einen Rücksetzeingang (R) zurückgesetzt werden und gibt den aktuellen Zählerstand über eine Terminalausgabe aus. Zusätzlich wird ein Ausgangssignal (Q) gesetzt, sobald der Zählerstand den vorgegebenen Sollwert (PV) erreicht oder überschreitet.

Verwendete Funktionsbausteine (FBs)

- **ALI_FB_CTU** (`adapter::iec61131::counters::ALI_FB_CTU`) Vorwärtszähler (Count Up) als Adapterbaustein.
  - Ereigniseingänge: – (wird über Adapterverbindungen gesteuert)
  - Daten: PV (Sollwert, LINT), CU (Zählimpuls), R (Rücksetzen)
  - Ausgänge: Q (bool), CV (aktueller Zählerstand, LINT)
- **ALI_LINT_TO_LI** (`adapter::conversion::unidirectional::ALI_LINT_TO_LI`) Konvertiert einen LINT-Wert in einen LI-Wert (LINT zu LINT?).
  - Parameter: `OUT` = `LINT#5` (Standardwert, wird beim Initialisieren gesetzt)
  - Ereigniseingang: `REQ` – löst die Konvertierung aus
  - Datenausgang: `ALI_OUT` (LI) wird mit `PV` des Zählers verbunden
- **Input_CU** (`logiBUS::io::DI::logiBUS_IXA`) Digitaler Eingang für den Zählimpuls (CU).
  - Parameter: `QI` = `TRUE`, `Input` = `Input_I1`
- **Input_R** (`logiBUS::io::DI::logiBUS_IXA`) Digitaler Eingang für das Rücksetzen (R).
  - Parameter: `QI` = `TRUE`, `Input` = `Input_I2`
- **Output_Q1** (`logiBUS::io::DQ::logiBUS_QXA`) Digitaler Ausgang, der den Zählerausgang Q (erreicht/überschritten) signalisiert.
  - Parameter: `QI` = `TRUE`, `Output` = `Output_Q1`

- **ALI_TO_AUDI** (`adapter::conversion::unidirectional::ALI_TO_AUDI`) Konvertiert einen ALI-Wert (hier den Zählerstand) in einen AUDI-Wert für die Terminalausgabe.
  - Hinweis: Die Konvertierung unterstützt keine negativen Zahlen (siehe Kommentar im Modell).
- **Q_NumericValue_AUDI** (`isobus::UT::Q::Q_NumericValue_AUDI`) Baustein zur Darstellung eines numerischen Werts auf dem Terminal (HMI).
  - Parameter: `u16ObjId` = `OutputNumber_N1`

Programmablauf und Verbindungen

Der Ablauf wird durch Ereignis- und Adapterverbindungen gesteuert:

1. **Initialisierung** Beim Start (Ereignis `INIT0` von `Input_R`) wird der Sollwert (PV) über den Baustein `ALI_LINT_TO_LI` gesetzt. Dieser liefert standardmäßig den Wert `LINT#5` als Sollwert.
2. **Zählbetrieb**
   - Jeder positive Flanke am Eingang `Input_CU` (verbunden mit dem Adaptereingang `CU` des Zählers) erhöht den internen Zählerstand um 1.
   - Ein Signal an `Input_R` (verbunden mit Adaptereingang `R`) setzt den Zähler zurück auf 0.
   - Der Ausgang `Q` des Zählers wird `TRUE`, sobald der Zählerstand größer oder gleich dem Sollwert (PV) ist. Dieses Signal wird an den digitalen Ausgang `Output_Q1` weitergegeben.
3. **Ausgabe des Zählerstands** Der aktuelle Zählerstand (CV) wird über die Adapterverbindung an `ALI_TO_AUDI` übergeben, dort in ein AUDI-Format konvertiert und schließlich an den Terminalbaustein `Q_NumericValue_AUDI` gesendet.
   - **Hinweis**: Die Konvertierung über `ALI_TO_AUDI` kann keine negativen Zahlen verarbeiten. Soll der Zählerstand auch negativ sein, müsste ein anderer Konvertierungsbaustein verwendet werden.
   - **Optimierungsvorschlag**: Um die Anzahl der Ereignisse zu reduzieren, könnte ein `AX_D_FF` (D-Flipflop) zwischengeschaltet werden (siehe Kommentar im Modell).

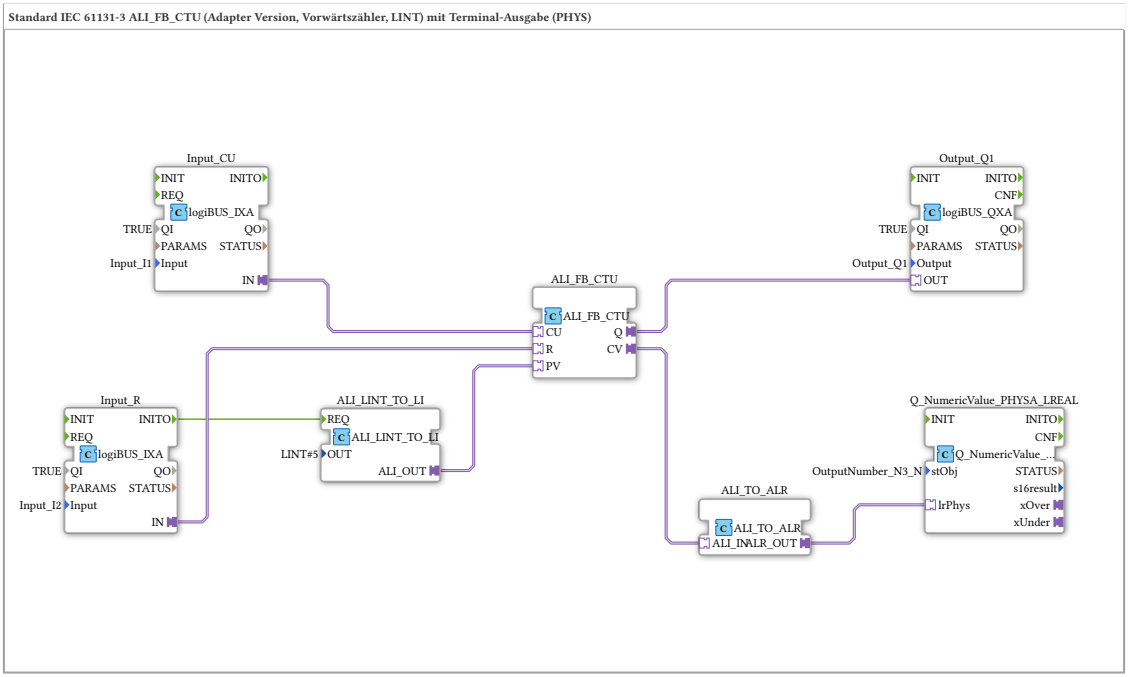
Zusammenfassung

Die Übung zeigt die praktische Anwendung eines IEC-61131-3-konformen Vorwärtszählers (Adapter-Version) mit Terminalausgabe. Der Zähler zählt Impulse auf einen einstellbaren Sollwert hoch, setzt einen Ausgang bei Erreichen des Sollwerts und gibt den Zählerstand auf einem Terminal aus. Die verwendeten Konvertierungsbausteine demonstrieren die Datenfluss- und Adapterkonzepte der 4diac-IDE.

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

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

2.1.2.493 Uebung_212b_ALR: Standard IEC 61131-3 ALI_FB_CTU (Adapter Version, Vorwärtszähler, LINT) mit Terminal-Ausgabe (PHYS)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung implementiert einen Vorwärtszähler (CTU) nach IEC 61131-3 als Adapter-Version. Der Zähler verwendet den Typ `ALI_FB_CTU` und kann auf einen vorgegebenen Preset-Wert (hier 5) voreingestellt werden. Der aktuelle Zählerstand wird über eine physikalische Terminalausgabe (`Q_NumericValue_PHYSA_LREAL`) auf einem Ausgabekanal ausgegeben. Zusätzlich wird ein digitaler Ausgang (`Output_Q1`) gesetzt, sobald der Zählwert den Preset-Wert erreicht oder überschreitet. Die Eingänge für das Zählsignal (CU) und den Reset (R) werden von digitalen Eingängen der logiBUS-Plattform gespeist. Ein Kommentar weist darauf hin, dass negative Zählerstände möglich sind und empfiehlt ggf. den Einbau eines AX_D_FF zur Reduzierung von Ereignissen.

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine

ALI_FB_CTU

- **Typ**: `adapter::iec61131::counters::ALI_FB_CTU`
- **Parameter**: Keine
- **Ereignis-Eingänge/-Ausgänge**: Keine direkten Event-Verbindungen; Ereignisse werden über die Adapter-Ports `CU`, `R`, `Q`, `CV` transportiert.
- **Daten-Eingänge/-Ausgänge**:
  - **Eingänge**: `CU` (Zählimpuls), `R` (Reset), `PV` (Preset-Wert, über Adapter von `ALI_LINT_TO_LI`)
  - **Ausgänge**: `Q` (Ziel erreicht), `CV` (aktueller Zählerstand, Typ LINT)
- **Funktionsweise**: Der Baustein ist ein Vorwärtszähler für LINT-Werte. Bei jedem steigenden Flanke am `CU`-Eingang wird der interne Zähler um 1 erhöht. Ein Signal am `R`-Eingang setzt den Zähler auf 0 zurück. Der Ausgang `Q` wird TRUE, sobald der aktuelle Zählerstand mindestens den Wert von `PV` erreicht. Der aktuelle Zählerstand ist am Ausgang `CV` verfügbar.

ALI_LINT_TO_LI

- **Typ**: `adapter::conversion::unidirectional::ALI_LINT_TO_LI`
- **Parameter**:
  - `OUT` = `LINT#5` (konstanter Preset-Wert)
- **Ereignis-Eingänge/-Ausgänge**:

  - **Ereignis-Eingang**: `REQ` (wird durch `INIT0` von `Input_R` getriggert)
- **Daten-Eingänge/-Ausgänge**:
  - **Ausgang**: `ALI_OUT` (liefert den konstanten Wert 5 als ALI-Signal an den `PV`-Eingang des Zählers)
- **Funktionsweise**: Dieser Baustein konvertiert einen konstanten LINT-Wert in das ALI-Format, das der Zähler als Preset-Eingang erwartet. Die Ausgabe wird bei einem Ereignis am `REQ`-Eingang aktualisiert (hier einmalig bei Initialisierung).

Input_CU

- **Typ**: `logiBUS::io::DI::logiBUS_IXA`
- **Parameter**:
  - `QI` = `TRUE` (Qualifier, immer aktiv)
  - `Input` = `Input_I1` (physikalischer DI-Kanal)
- **Ereignis-Eingänge/-Ausgänge**: Keine direkten Event-Verbindungen
- **Daten-Eingänge/-Ausgänge**:
  - **Ausgang**: `IN` (Adapter, liefert das Digitaleingangssignal an den `CU`-Eingang des Zählers)
- **Funktionsweise**: Liest den Zustand des digitalen Eingangs `Input_I1` und stellt ihn als Adapter-Signal bereit.

Input_R

- **Typ**: `logiBUS::io::DI::logiBUS_IXA`
- **Parameter**:
  - `QI` = `TRUE`
  - `Input` = `Input_I2`
- **Ereignis-Eingänge/-Ausgänge**:
  - **Ereignis-Ausgang**: `INIT0` (wird beim ersten Einschalten aktiviert und triggert `ALI_LINT_TO_LI.REQ`)
- **Daten-Eingänge/-Ausgänge**:
  - **Ausgang**: `IN` (Adapter, liefert das Resetsignal an den `R`-Eingang des Zählers)
- **Funktionsweise**: Liest den Zustand des digitalen Eingangs `Input_I2` und stellt ihn als Adapter-Signal für den Zähler-Reset bereit. Der `INIT0`-Ereignisausgang wird für die einmalige Initialisierung des Preset-Wertes genutzt.

Output_Q1

- **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
- **Parameter**:
  - `QI` = `TRUE`
  - `Output` = `Output_Q1` (physikalischer DO-Kanal)
- **Daten-Eingänge/-Ausgänge**:
  - **Eingang**: `OUT` (Adapter, erhält das `Q`-Signal des Zählers)
- **Funktionsweise**: Setzt den digitalen Ausgang `Output_Q1` auf den Wert des Zählerausgangs `Q`.

ALI_TO_ALR

- **Typ**: `adapter::conversion::unidirectional::ALI_TO_ALR`
- **Parameter**: Keine
- **Daten-Eingänge/-Ausgänge**:
  - **Eingang**: `ALI_IN` (erhält den aktuellen Zählerstand `CV` vom Zähler)
  - **Ausgang**: `ALR_OUT` (liefert den Wert als ALR-Signal an die Terminalausgabe)
- **Funktionsweise**: Wandelt das ALI-Signal (LINT) in ein ALR-Signal (LREAL) um, das für die physikalische Ausgabe benötigt wird.

Q_NumericValue_PHYSA_LREAL

- **Typ**: `isobus::UT::Q::Q_NumericValue_PHYSA_LREAL`
- **Parameter**:
  - `stObj` = `OutputNumber_N3` (Referenz auf das Terminalausgabeobjekt)
- **Daten-Eingänge/-Ausgänge**:
  - **Eingang**: `lrPhys` (erhält das ALR-Signal von `ALI_TO_ALR`)
- **Funktionsweise**: Gibt den übergebenen LREAL-Wert als numerischen Wert auf der physikalischen Terminalausgabe `OutputNumber_N3` aus.

Programmablauf und Verbindungen

Die Übung realisiert einen vorwärtszählenden Zähler mit Terminalausgabe. Die Verbindungen sind wie folgt aufgebaut:

1. **Initialisierung**: Beim Start der SPS wird das INITO-Ereignis von `Input_R` ausgelöst. Dieses triggert `ALI_LINT_TO_LI` und setzt den Preset-Wert des Zählers auf `LINT#5`. Der Zähler ist damit für den Zielwert 5 konfiguriert.
2. **Zählbetrieb**: Der digitale Eingang `Input_I1` (Taster oder Sensor) wird über `Input_CU` an den `CU`-Eingang des Zählers geführt. Jede steigende Flanke erhöht den internen Zählerstand. Der digitale Eingang `Input_I2` wird über `Input_R` an den `R`-Eingang geführt. Ein Signal setzt den Zähler auf `0` zurück.
3. **Ausgänge**:
   - Der Ausgang `Q` des Zählers wird über `Output_Q1` auf den digitalen Ausgang `Output_Q1` gegeben. Dieser wird TRUE, sobald der Zählerstand >= 5 ist.
   - Der aktuelle Zählerstand (`CV`) wird über `ALI_TO_ALR` in einen LREAL-Wert konvertiert und über `Q_NumericValue_PHYSA_LREAL` auf dem Terminal `OutputNumber_N3` ausgegeben. Dadurch kann der Zählerstand in einer Visualisierung oder auf einem Display angezeigt werden.
4. **Besonderheiten**: Ein Kommentar im Netzwerk weist darauf hin, dass negative Zählerstände möglich sind (z. B. durch Überlauf oder fehlerhafte Verwendung). Zudem wird empfohlen, bei Bedarf einen AX_D_FF-Baustein einzufügen, um die Anzahl der Ereignisse (insbesondere bei schnellen Zählimpulsen) zu reduzieren und die Systemlast zu verringern.

Lernziele: Verständnis des IEC 61131-3 Zählers (CTU) in der Adapter-Version, Umgang mit Konstanten und Konvertierungsbausteinen, Anbindung digitaler Ein- und Ausgänge sowie physikalischer Terminalausgaben.

Schwierigkeitsgrad: Mittel – Grundkenntnisse der 4diac-IDE und des logiBUS-Systems werden vorausgesetzt.

Benötigte Vorkenntnisse: Umgang mit Funktionsbausteinen, Adapterverbindungen und Ereignissteuerung in 4diac.

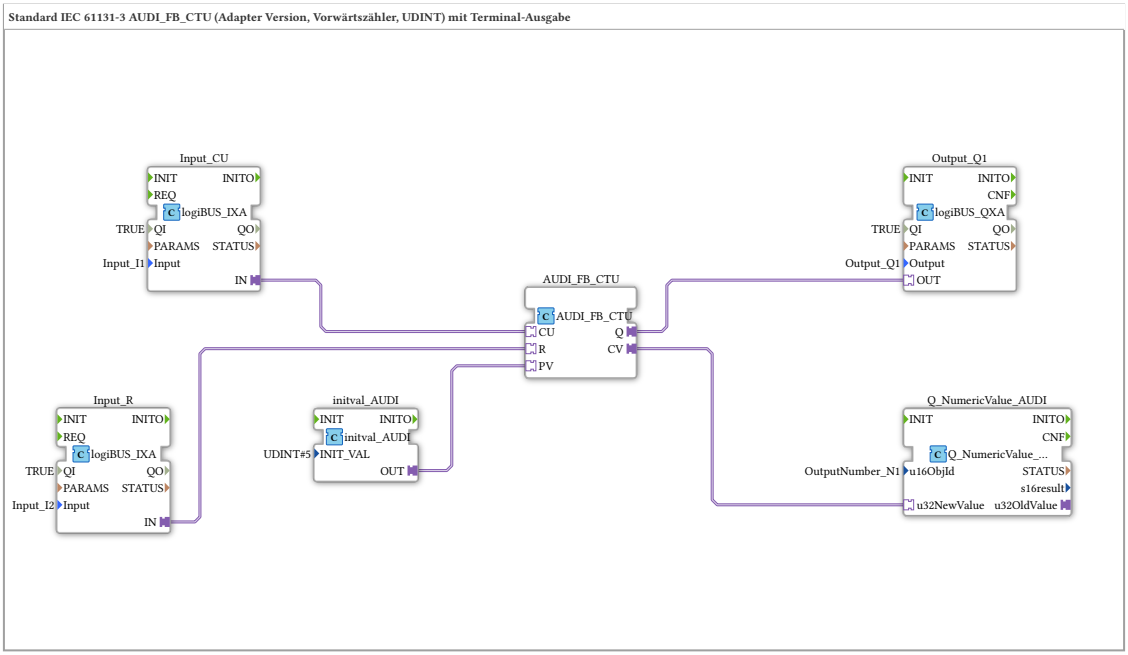
Zusammenfassung

Die Übung `Uebung_212b_ALR` demonstriert einen vollständig konfigurierten Vorwärtszähler mit festem Preset-Wert und physikalischer Ausgabe. Sie kombiniert digitale Ein-/Ausgänge, einen Zählbaustein, Konvertierungsbausteine und eine Terminalausgabe zu einem funktionsfähigen Automatisierungsbeispiel. Die Kommentare geben praktische Hinweise zur Optimierung (Ereignisreduzierung) und weisen auf mögliche Randbedingungen (negative Werte) hin.

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

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

2.1.2.494 Uebung_213_AUDI: Standard IEC 61131-3 AUDI_FB_CTU (Adapter-Version, Vorwärtszähler, UDINT) mit Terminal-Ausgabe



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung zeigt einen Vorwärtszähler (CTU) nach IEC 61131-3 als Adapter-Version für den Datentyp UDINT. Der Zähler wird über zwei digitale Eingänge bedient (Zählen und Reset), der feste Vorgabewert `5` kommt aus einem `initval_AUDI`-Konstantenbaustein. Der Zählerstand wird auf dem ISOBUS-Terminal angezeigt, das Erreichen des Vorgabewerts auf `Q1` ausgegeben.

Verwendete Funktionsbausteine (FBs)

- **Input_CU**: Zähleingang
  - **Typ**: `logiBUS::io::DI::logiBUS_IXA`
  - **Parameter**: `QI = TRUE`, `Input = Input_I1`
  - **Erklärung**: Stellt den Zustand von `I1` als Adapterausgang `IN` bereit.
- **Input_R**: Reset-Eingang
  - **Typ**: `logiBUS::io::DI::logiBUS_IXA`
  - **Parameter**: `QI = TRUE`, `Input = Input_I2`
  - **Erklärung**: Stellt den Zustand von `I2` als Adapterausgang `IN` bereit.
- **initval_AUDI**: Konstante für den Vorgabewert
  - **Typ**: `adapter::types::unidirectional::AUDI::initval::initval_AUDI`
  - **Parameter**: `INIT_VAL = 5`
  - **Erklärung**: Liefert den Vorgabewert `5` als `AUDI`-Adapterwert an `AUDI_FB_CTU.PV`.
- **AUDI_FB_CTU**: Vorwärtszähler (UDINT, Adapter-Version)
  - **Typ**: `adapter::iec61131::counters::AUDI_FB_CTU`
  - **Parameter**: keine
  - **Erklärung**: Zählt über den Adapter-Socket `CU` aufwärts, `R` setzt den Zähler zurück, `PV` ist der Vorgabewert. Der Adapterausgang `Q` ist `TRUE`, wenn der Zählerstand `CV` den Vorgabewert `PV` erreicht oder überschreitet; `CV` liefert den aktuellen Zählerstand.
- **Output_Q1**: Digitalausgang
  - **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
  - **Parameter**: `QI = TRUE`, `Output = Output_Q1`
  - **Erklärung**: Gibt den Zählerausgang `Q` am physischen Ausgang `Q1` aus.
- **Q_NumericValue_AUDI**: Anzeige auf dem ISOBUS-Terminal
  - **Typ**: `isobus::UT::Q::Q_NumericValue_AUDI`
  - **Parameter**: `u16ObjId = OutputNumber_N1`

– **Erklärung**: Empfängt den Zählerstand über den `AUDI`-Adapter `u32NewValue` und zeigt ihn im Objekt `OutputNumber_N1` an.

Programmablauf und Verbindungen

1. `Input_CU.IN` ⟶ `AUDI_FB_CTU.CU` (AdapterConnection): Der Zustand von `I1` ist der Zähleingang.
2. `Input_R.IN` ⟶ `AUDI_FB_CTU.R` (AdapterConnection): Der Zustand von `I2` setzt den Zähler zurück.
3. `initval_AUDI.OUT` ⟶ `AUDI_FB_CTU.PV` (AdapterConnection): Der Vorgabewert `5` wird als Preset übergeben.
4. `AUDI_FB_CTU.Q` ⟶ `Output_Q1.OUT` (AdapterConnection): Der Zählerausgang liegt an `Q1`.
5. `AUDI_FB_CTU.CV` ⟶ `Q_NumericValue_AUDI.u32NewValue` (AdapterConnection): Der Zählerstand wird auf dem Terminal angezeigt.

Verhalten: Jede steigende Flanke an `I1` erhöht den Zählerstand; `Q1` wird `TRUE`, sobald der Zählerstand `5` erreicht oder überschreitet. `I2` setzt den Zähler zurück. Der Zählerstand erscheint in `OutputNumber_N1`. Laut Netzwerk-Kommentar kann hier gegebenenfalls ein `AX_D_FF` eingebaut werden, damit die Events reduziert werden.

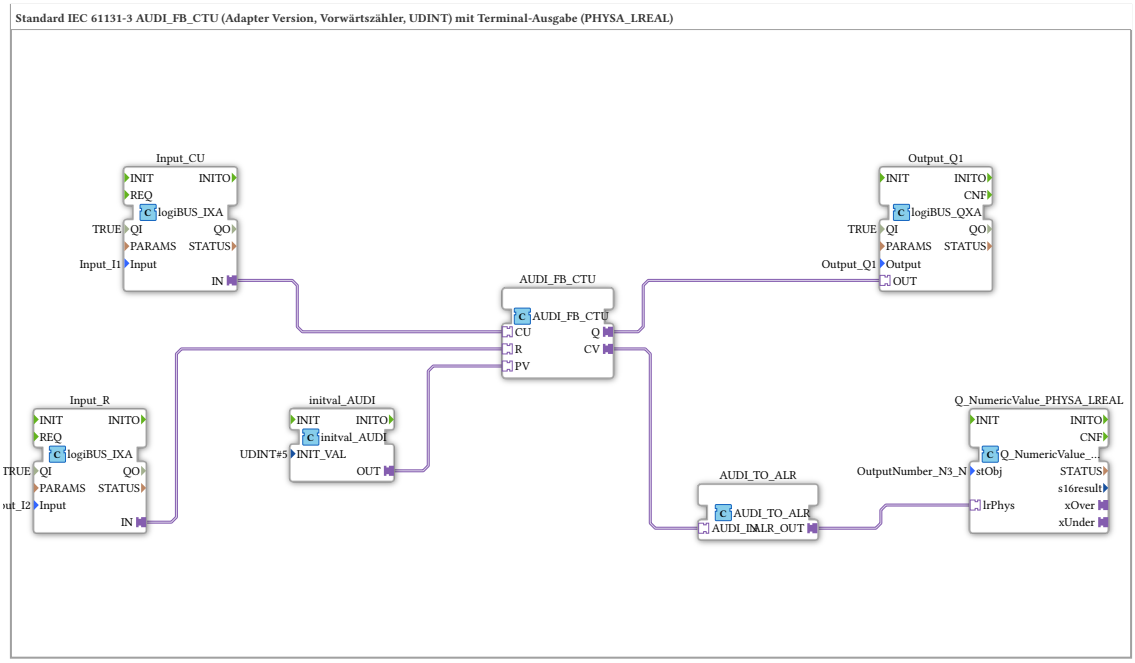
Zusammenfassung

Die Übung zeigt die adapterbasierte Anbindung eines IEC-61131-3-Zählers: Ein- und Ausgänge, Vorgabewert und Anzeige sind rein über Adapterverbindungen verschaltet, ohne zusätzliche Ereignisverbindungen.

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

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

2.1.2.495 Uebung_213b_ALR: Standard IEC 61131-3 AUDI_FB_CTU (Adapter-Version, Vorwärtszähler, UDINT) mit Terminal-Ausgabe (PHYSA_LREAL)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung realisiert einen Vorwärtszähler (CTU) nach IEC 61131-3 mit dem Zählbereich `UDINT` als Adapter-Version. Taster `I1` zählt hoch (`CU`), `I2` setzt zurück (`R`). Der Vorgabewert `PV` ist über `initval_AUDI` fest auf 5 gesetzt, sodass der Ausgang `Q` bei Erreichen von 5 aktiv wird und `Q1` schaltet. Der Zählerstand `CV` wird in einen `LREAL`-Wert gewandelt und am Terminal-Objekt `OutputNumber_N3_N` ausgegeben.

Verwendete Funktionsbausteine (FBs)

- **Input_CU**: Zählimpulse
  - **Typ**: `logiBUS::io::DI::logiBUS_IXA`
  - **Parameter**: `QI = TRUE`, `Input = Input_I1`
  - **Erklärung**: Stellt den Zustand von `I1` als Adapter `IN` bereit; geht an `AUDI_FB_CTU.CU`.
- **Input_R**: Rücksetzen
  - **Typ**: `logiBUS::io::DI::logiBUS_IXA`
  - **Parameter**: `QI = TRUE`, `Input = Input_I2`
  - **Erklärung**: Stellt den Zustand von `I2` als Adapter `IN` bereit; geht an `AUDI_FB_CTU.R`.
- **initval_AUDI**: Vorgabewert
  - **Typ**: `adapter::types::unidirectional::AUDI::initval::initval_AUDI`
  - **Parameter**: `INIT_VAL = 5`
  - **Erklärung**: Liefert den Wert 5 als Vorgabewert `PV` an den Zähler.
- **AUDI_FB_CTU**: Vorwärtszähler
  - **Typ**: `adapter::iec61131::counters::AUDI_FB_CTU`
  - **Parameter**: keine
  - **Erklärung**: Zählereingang `CU`, Rücksetzeingang `R`, Vorgabewert `PV`; Ausgänge `Q` (aktiv, wenn `CV` ≥ `PV`) und `CV` (aktueller Zählerstand).
- **Output_Q1**: logiBUS Digitalausgang
  - **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
  - **Parameter**: `QI = TRUE`, `Output = Output_Q1`
  - **Erklärung**: Gibt `AUDI_FB_CTU.Q` am physischen Ausgang `Q1` aus.
- **AUDI_TO_ALR**: Konvertierung `AUDI` nach `ALR`
  - **Typ**: `adapter::conversion::unidirectional::AUDI_TO_ALR`
  - **Parameter**: keine
  - **Erklärung**: Wandelt den Zählerstand (`AUDI_IN`) in einen `LREAL`-Adapterwert (`ALR_OUT`). Laut Kommentar im Netzwerk sind hier negative Werte möglich.
- **Q_NumericValue_PHYSA_LREAL**: Terminal-Ausgabe
  - **Typ**: `isobus::UT::Q::Q_NumericValue_PHYSA_LREAL`
  - **Parameter**: `stObj = OutputNumber_N3_N`
  - **Erklärung**: Zeigt den `LREAL`-Wert aus `lrPhys` am Terminal-Objekt `OutputNumber_N3_N` an.

Programmablauf und Verbindungen

1. `Input_CU.IN` ⟶ `AUDI_FB_CTU.CU` und `Input_R.IN` ⟶ `AUDI_FB_CTU.R` (AdapterConnections): `I1` zählt, `I2` setzt zurück.
2. `initval_AUDI.OUT` ⟶ `AUDI_FB_CTU.PV` (AdapterConnection): Vorgabewert 5.
3. `AUDI_FB_CTU.Q` ⟶ `Output_Q1.OUT` (AdapterConnection): `Q1` ist aktiv, wenn der Zählerstand den Vorgabewert erreicht oder überschreitet.
4. `AUDI_FB_CTU.CV` ⟶ `AUDI_TO_ALR.AUDI_IN` und `AUDI_TO_ALR.ALR_OUT` ⟶ `Q_NumericValue_PHYSA_LREAL.lrPhys` (AdapterConnections): Der Zählerstand wird als `LREAL` am Terminal angezeigt.

Hinweis aus dem Netzwerk: Hier kann gegebenenfalls ein `AX_D_FF` eingebaut werden, damit die Events reduziert werden.

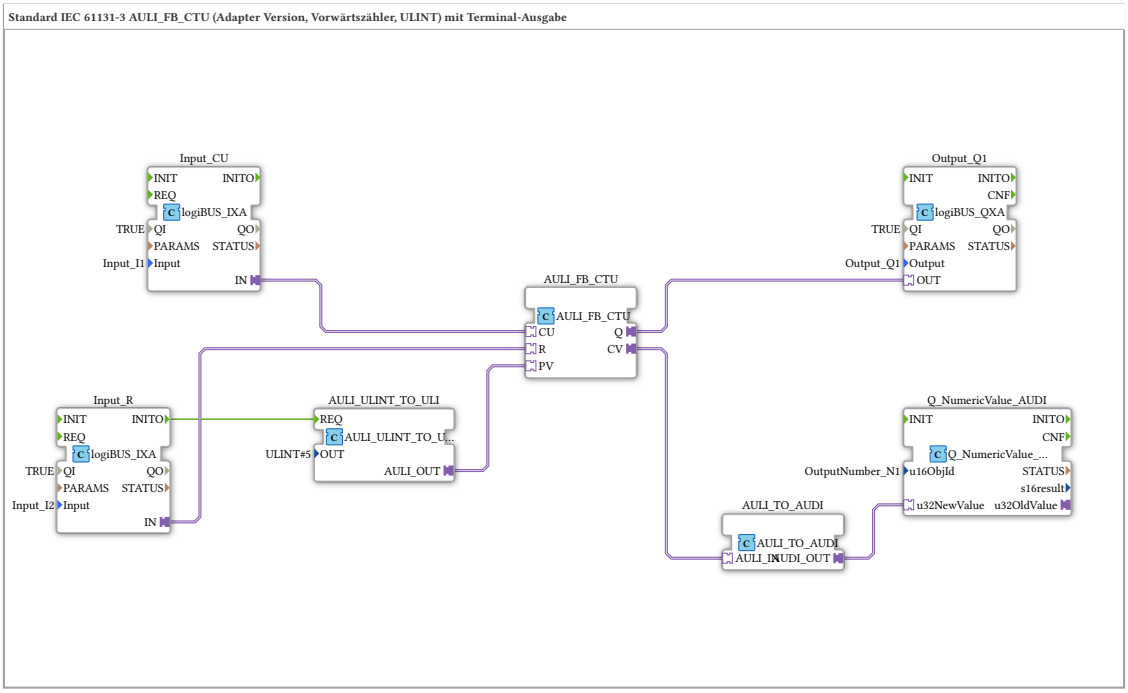
Zusammenfassung

Die Übung 213b_ALR zeigt einen adapterbasierten IEC-Vorwärtszähler mit festem Vorgabewert 5, dessen Zählerstand über `AUDI_TO_ALR` auf einem Terminal ausgegeben wird. Alle Verbindungen sind Adapterverbindungen.

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

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

2.1.2.496 Uebung_214_AULI: Standard IEC 61131-3 AULI_FB_CTU (Adapter Version, Vorwärtszähler, ULINT) mit Terminal-Ausgabe



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung implementiert einen Vorwärtszähler gemäß IEC 61131-3 (Typ **AULI_FB_CTU**) als Adapter-Version. Der Zähler arbeitet mit dem Datentyp **ULINT** und gibt seinen aktuellen Zählerstand sowie den Überlauf (Q) an eine Terminal-Ausgabe (Numerische Anzeige) und einen digitalen Ausgang weiter. Zusätzlich wird der Startwert (PV) über einen Konvertierungsbaustein gesetzt. Die Übung demonstriert den Einsatz von Adapter-FBs, I/O-Anbindung und Datenkonvertierung.

Verwendete Funktionsbausteine (FBs)

Die SubApp besteht aus mehreren Funktionsbausteinen, die im Folgenden beschrieben werden.

Sub-Baustein: AULI_FB_CTU

- **Typ**: `adapter::iec61131::counters::AULI_FB_CTU`
- **Parameter**: Keine (werden über Adapterverbindungen gesetzt)
- **Funktionsweise**: Vorwärtszähler für ULINT-Werte. Er zählt bei jedem positiven Flanke am Eingang **CU** (Count Up) hoch. Der Eingang **R** setzt den Zähler zurück. Der aktuelle Zählerwert wird am Adapterausgang **CV** ausgegeben, der Überlauf (Zählerstand ≥ PV) am Ausgang **Q**.

Sub-Baustein: AULI_ULINT_TO_ULI

- **Typ**: `adapter::conversion::unidirectional::AULI_ULINT_TO_ULI`
- **Parameter**:
  - `OUT` = `ULINT#5` (Voreinstellung des Startwerts)
- **Funktionsweise**: Konvertiert einen ULINT-Wert (hier konstant 5) in einen ULI-Ausgangswert, der dem Zähler als **PV** (Preset Value) zugeführt wird. Der Baustein wird beim Start (INITO des Reseteingangs) aktiviert.

Sub-Baustein: Input_CU

- **Typ**: `logiBUS::io::DI::logiBUS_IXA`
- **Parameter**:
  - `QI` = `TRUE`
  - `Input` = `Input_I1` (physikalischer Eingang 1)

- **Funktionsweise**: Digitaler Eingang, der die Zählimpulse (CU) liefert. Der Baustein stellt den Adapterausgang **IN** zur Verfügung.

Sub-Baustein: Input_R

- **Typ**: `logiBUS::io::DI::logiBUS_IXA`
- **Parameter**:
  - `QI` = `TRUE`
  - `Input` = `Input_I2` (physikalischer Eingang 2)
- **Funktionsweise**: Digitaler Eingang zum Zurücksetzen des Zählers (R). Sein Ereignisausgang **INITO** wird genutzt, um beim Start die Initialisierung des PV-Wertes auszulösen.

Sub-Baustein: Output_Q1

- **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
- **Parameter**:
  - `QI` = `TRUE`
  - `Output` = `Output_Q1` (physikalischer Ausgang 1)
- **Funktionsweise**: Digitaler Ausgang, der den Überlauf (Q) des Zählers anzeigt.

Sub-Baustein: AULI_TO_AUDI

- **Typ**: `adapter::conversion::unidirectional::AULI_TO_AUDI`
- **Parameter**: Keine
- **Funktionsweise**: Konvertiert den Zählerwert vom Typ ULINT (AULI) in einen numerischen Wert (AUDI) für die Terminal-Ausgabe.

Sub-Baustein: Q_NumericValue_AUDI

- **Typ**: `isobus::UT::Q::Q_NumericValue_AUDI`
- **Parameter**:
  - `u16ObjId` = `OutputNumber_N1` (Objekt-ID der numerischen Anzeige)
- **Funktionsweise**: Gibt den konvertierten Zählerwert auf einem Terminal (numerische Anzeige) aus.

Programmablauf und Verbindungen

Der Ablauf der Übung ist wie folgt:

1. **Initialisierung**: Beim Start (Ereignis **INITO** des Eingangsbausteins `Input_R`) wird der Baustein `AULI_ULINT_TO_ULI` getriggert, der den konstanten Wert `ULINT#5` in einen ULI-Wert umwandelt und dem Zähler als **PV** (Preset Value) zuweist.
2. **Zählvorgang**: Jeder positive Flanke am physikalischen Eingang `Input_I1` (verbunden mit `Input_CU`) löst am Zähler **CU** einen Zählschritt aus. Der Zähler erhöht seinen internen Wert.
3. **Rücksetzen**: Eine positive Flanke am Eingang `Input_I2` (verbunden mit `Input_R`) setzt den Zähler auf 0 zurück.
4. **Ausgabe**:
   - Der Überlaufausgang **Q** des Zählers wird an den digitalen Ausgang `Output_Q1` weitergegeben.
   - Der aktuelle Zählerwert **CV** wird über die Konvertierungskette (`AULI_TO_AUDI` → `Q_NumericValue_AUDI`) auf einem Terminal (numerische Anzeige mit Objekt-ID `OutputNumber_N1`) dargestellt.

Hinweis: Ein Kommentar im Netzwerk schlägt vor, bei Bedarf einen **AX_D_FF** (Flip-Flop) einzufügen, um die Ereignisrate zu reduzieren.

Die Verbindungen im Überblick:

- **Event-Verbindung**: `Input_R.INITO` ⟶ `AULI_ULINT_TO_ULI.REQ`
- **Adapterverbindungen**:
  - `Input_CU.IN` ⟶ `AULI_FB_CTU.CU`
  - `Input_R.IN` ⟶ `AULI_FB_CTU.R`
  - `AULI_FB_CTU.Q` ⟶ `Output_Q1.OUT`
  - `AULI_FB_CTU.CV` ⟶ `AULI_TO_AUDI.AULI_IN`
  - `AULI_TO_AUDI.AUDI_OUT` ⟶ `Q_NumericValue_AUDI.u32NewValue`
  - `AULI_ULINT_TO_ULI.AULI_OUT` ⟶ `AULI_FB_CTU.PV`

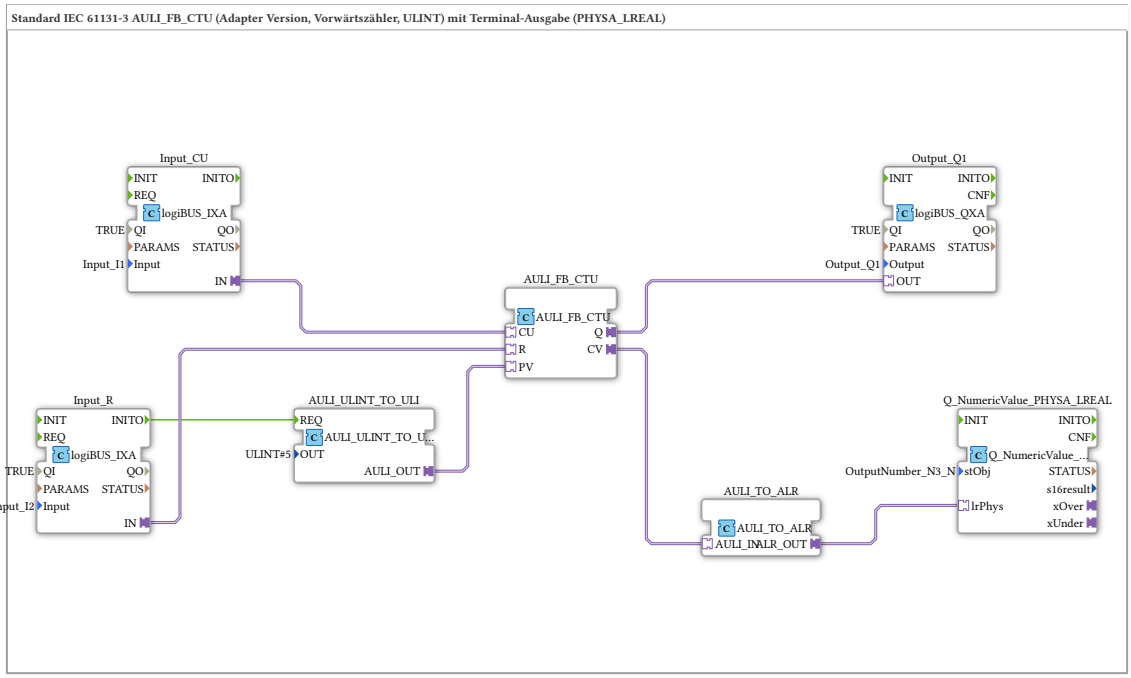
Zusammenfassung

Die Übung **Uebung_214_AULI** vermittelt den Umgang mit dem IEC-61131-3-konformen Vorwärtszähler **AULI_FB_CTU** in einer Adapter-basierten Umgebung. Sie zeigt, wie ein Zähler über digitale Eingänge gesteuert, sein Wert über Konvertierungsbausteine aufbereitet und sowohl auf einem digitalen Ausgang als auch auf einem Terminal ausgegeben wird. Die Integration von Initialisierungslogik und die flexible Anbindung über Adapter machen diese Übung zu einer soliden Grundlage für komplexere Automatisierungsaufgaben mit 4diac.

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

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

2.1.2.497 Uebung_214b_ALR: Standard IEC 61131-3 AULI_FB_CTU (Adapter Version, Vorwärtszähler, ULINT) mit Terminal-Ausgabe (PHYSA_LREAL)



• — ○ • — ○ • — ○ •

Einleitung

Diese Übung implementiert einen Vorwärtszähler (CTU) nach IEC 61131-3 als Adapter-Version. Der Zähler arbeitet mit dem Datentyp `ULINT`. Der aktuelle Zählwert wird über einen physikalischen Ausgang (`PHYSA_LREAL`) an ein Terminal ausgegeben. Der Preset-Wert wird initial auf 5 gesetzt.

Verwendete Funktionsbausteine (FBs)

AULI_FB_CTU

- **Typ**: `adapter::iec61131::counters::AULI_FB_CTU`
- **Parameter**: keine
- **Ereigniseingänge**: (implizit über Adapterverbindungen)
- **Ereignisausgänge**: (implizit)
- **Dateneingänge**:
  - `CU` (Count Up) – Takteingang zum Hochzählen
  - `R` (Reset) – setzt Zähler zurück
  - `PV` (Preset Value) – Vorgabewert, initial `ULINT#5`
- **Datenausgänge**:
  - `Q` (Ausgang) – wird `TRUE`, wenn `CV >= PV`
  - `CV` (aktueller Zählwert) – `ULINT`

AULI_ULINT_TO_ULI

- **Typ**: `adapter::conversion::unidirectional::AULI_ULINT_TO_ULI`
- **Parameter**: `OUT = ULINT#5` (statischer Wert, der als Preset dient)
- **Ereigniseingänge**:
  - `REQ` (Request) – triggert Umwandlung (verbunden mit `Input_R.INITO`)
- **Datenausgänge**:
  - `AULI_OUT` – liefert den umgewandelten Wert an `AULI_FB_CTU.PV`

Input_CU

- **Typ**: `logiBUS::io::DI::logiBUS_IXA`
- **Parameter**:
  - `QI = TRUE` (Aktivierung)
  - `Input = Input_I1` (physischer Digitaleingang 1)
- **Datenausgänge**:
  - `IN` – Adapterausgang, verbunden mit `AULI_FB_CTU.CU`

Input_R

- **Typ**: `logiBUS::io::DI::logiBUS_IXA`
- **Parameter**:
  - `QI = TRUE`
  - `Input = Input_I2` (physischer Digitaleingang 2)
- **Datenausgänge**:
  - `IN` – Adapterausgang, verbunden mit `AULI_FB_CTU.R`
- **Ereignisausgänge**:
  - `INITO` – Initialisierungsereignis, verbunden mit `AULI_ULINT_TO_ULI.REQ`

Output_Q1

- **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
- **Parameter**:
  - `QI = TRUE`
  - `Output = Output_Q1` (physischer Digitalausgang 1)
- **Dateneingänge**:
  - `OUT` – Adaptereingang, verbunden mit `AULI_FB_CTU.Q`

AULI_TO_AUDI (Instanzname `AULI_TO_AUDI`)

- **Typ**: `adapter::conversion::unidirectional::AULI_TO_ALR`
- **Parameter**: keine
- **Dateneingänge**:
  - `AULI_IN` – erhält den Zählwert (`AULI_FB_CTU.CV`)
- **Datenausgänge**:
  - `ALR_OUT` – liefert den Zählwert als `LREAL` an `Q_NumericValue_PHYSA_LREAL`

Q_NumericValue_PHYSA_LREAL

- **Typ**: `isobus::UT::Q::Q_NumericValue_PHYSA_LREAL`
- **Parameter**:
  - `stObj = OutputNumber_N3` (Referenz auf das Terminal-Ausgabeobjekt)
- **Dateneingänge**:
  - `lrPhys` – physikalischer `LREAL`-Wert, verbunden mit `AULI_TO_AUDI.ALR_OUT`

Programmablauf und Verbindungen

1. **Initialisierung**
   Beim Hochfahren der Steuerung löst `Input_R.INITO` ein Ereignis aus, das den Konverter `AULI_ULINT_TO_ULI.REQ` triggert. Dieser wandelt den statischen Wert `ULINT#5` um und übergibt ihn als Preset (`PV`) an den Zähler `AULI_FB_CTU`.
2. **Zählvorgang**
   - Jede steigende Flanke am Digitaleingang `Input_I1` (verbunden mit `Input_CU`) erhöht den Zähler `AULI_FB_CTU` um 1.
   - Der Zähler gibt den aktuellen Zählwert (`CV`) als `ULINT` aus.

- Wenn `CV >= PV` (hier ≥ 5), wird der Ausgang `Q` auf `TRUE` gesetzt. Dieser schaltet den Digitalausgang `Output_Q1` ein.

3. **Reset**
   Ein Signal am Digitaleingang `Input_I2` (verbunden mit `Input_R`) setzt den Zähler auf 0 zurück.
4. **Ausgabe auf Terminal**
   - Der Zählwert (`CV`) wird durch den Konverter `AULI_TO_AUDI` vom Typ `AULI_TO_ALR` in eine Gleitkommazahl (`LREAL`) umgewandelt.
   - Diese wird an den Funktionsbaustein `Q_NumericValue_PHYSA_LREAL` übergeben, der den Wert auf dem Terminal (Objekt `OutputNumber_N3`) anzeigt.

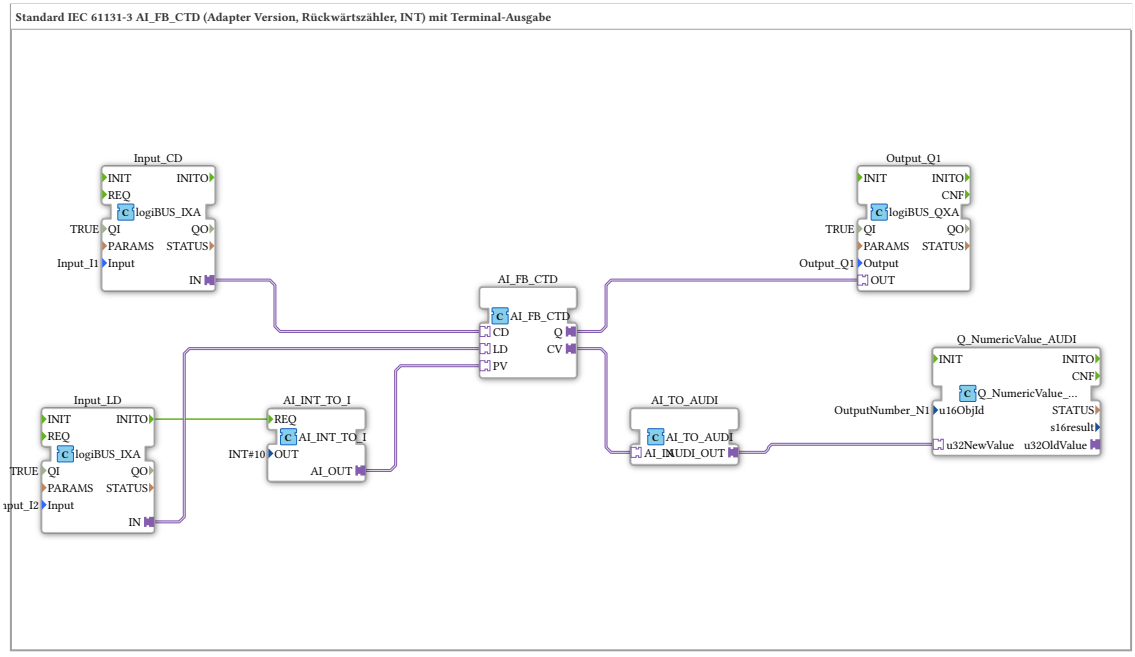
Hinweise:

- Negative Werte sind durch den verwendeten Datentyp `LREAL` möglich (siehe Kommentar im Netzwerk).
- Zur Reduzierung der Ereignisrate kann bei hohen Zählfrequenzen ein AX_D_FF (Abfrageverzögerung) vorgeschaltet werden (siehe Kommentar).

Zusammenfassung

Die Übung demonstriert die Anwendung eines IEC 61131-3 Vorwärtszählers als Adapter-FB mit `ULINT`-Datentyp. Der Zähler wird über digitale Eingänge gesteuert, der Ausgang schaltet einen Digitalausgang, und der aktuelle Zählwert wird über eine Typkonvertierung auf einem Terminal als `LREAL` ausgegeben. Der Preset-Wert wird initial auf 5 gesetzt.

2.1.2.498 Uebung_215_AI: Standard IEC 61131-3 AI_FB_CTD (Adapter Version, Rückwärtszähler, INT) mit Terminal-Ausgabe



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung implementiert einen Rückwärtszähler (Down-Counter) nach IEC 61131-3 mit Adapter-Schnittstelle (AI_FB_CTD) und gibt den aktuellen Zählerwert auf einem Terminal aus. Der Zähler wird über digitale Eingänge dekrementiert und geladen.

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: AI_FB_CTD

- **Typ**: adapter::iec61131::counters::AI_FB_CTD
- **Verwendete interne FBs**: Keine

- **Parameter**: Keine
- **Ereignisausgang/-eingang**: CD (Count Down), LD (Load), R (Reset); Q (Überlaufausgang)
- **Datenausgang/-eingang**: CV (aktueller Zählerwert, INT), PV (Preset Value, INT als Eingang über Adapter)
- **Funktionsweise**: Der Rückwärtszähler zählt bei jedem Ereignis an CD den Wert von PV herunter. Bei einem Ereignis an LD wird der Zähler auf den Wert von PV gesetzt. Der Ausgang Q wird TRUE, wenn der Zählerwert 0 erreicht oder unterschreitet.

Sub-Bausteine: AI_INT_TO_I

- **Typ**: adapter::conversion::unidirectional::AI_INT_TO_I
- **Verwendete interne FBs**: Keine
- **Parameter**: OUT = INT#10
- **Ereignisausgang/-eingang**: REQ (Eingang), CNF (Ausgang)
- **Datenausgang/-eingang**: AI_OUT (INT)
- **Funktionsweise**: Dieser Baustein stellt einen konstanten Integer-Wert (hier 10) bereit, der als Preset Value (PV) für den Zähler verwendet wird. Er wird durch das INITO-Ereignis des Load-Inputs getriggert.

Sub-Bausteine: Input_CD (Count Down-Eingang)

- **Typ**: logiBUS::io::DI::logiBUS_IXA
- **Verwendete interne FBs**: Keine
- **Parameter**: QI = TRUE, Input = Input_I1
- **Ereignisausgang/-eingang**: INITO (Initialisierung), IN (Ereignisausgang bei Flanke)
- **Datenausgang/-eingang**: Keine
- **Funktionsweise**: Liest den digitalen Eingang I1 des logiBUS-Systems und gibt bei einer positiven Flanke ein Ereignis auf dem Adapter-Ausgang aus. Dieses Ereignis triggert den CD-Eingang des Zählers.

Sub-Bausteine: Input_LD (Load-Eingang)

- **Typ**: logiBUS::io::DI::logiBUS_IXA
- **Verwendete interne FBs**: Keine
- **Parameter**: QI = TRUE, Input = Input_I2
- **Ereignisausgang/-eingang**: INITO (Initialisierung), IN (Ereignisausgang bei Flanke)
- **Datenausgang/-eingang**: Keine
- **Funktionsweise**: Liest den digitalen Eingang I2 und gibt bei einer positiven Flanke ein Ereignis aus. Dieses Ereignis triggert das LD-Ereignis am Zähler. Gleichzeitig wird über INITO die Initialisierung des PV-Werts angestoßen.

Sub-Bausteine: Output_Q1

- **Typ**: logiBUS::io::DQ::logiBUS_QXA
- **Verwendete interne FBs**: Keine
- **Parameter**: QI = TRUE, Output = Output_Q1
- **Ereignisausgang/-eingang**: OUT (Ereigniseingang), CNF (Bestätigung)
- **Datenausgang/-eingang**: Keine
- **Funktionsweise**: Übernimmt den Zählerausgang Q (über Adapter) und gibt ihn als digitalen Ausgang Q1 des logiBUS aus. Sobald der Zähler Null erreicht, wird Q1 aktiv.

Sub-Bausteine: AI_TO_AUDI

- **Typ**: adapter::conversion::unidirectional::AI_TO_AUDI
- **Verwendete interne FBs**: Keine
- **Parameter**: Keine
- **Ereignisausgang/-eingang**: REQ (Eingang), CNF (Ausgang)
- **Datenausgang/-eingang**: AI_IN (INT), AUDI_OUT (AUDI)
- **Funktionsweise**: Konvertiert den Integer-Zählerwert (CV) in das für das Terminal erforderliche AUDI-Format. Hinweis: Dieser Baustein unterstützt keine negativen Zahlen, was bei einem Rückwärtszähler problematisch sein kann.

Sub-Bausteine: Q_NumericValue_AUDI

- **Typ**: isobus::UT::Q::Q_NumericValue_AUDI
- **Verwendete interne FBs**: Keine
- **Parameter**: u16ObjId = OutputNumber_N1

- **Ereignisausgang/-eingang**: IN (Eingang für neuen Wert)
- **Datenausgang/-eingang**: u32NewValue (AUDI)
- **Funktionsweise**: Empfängt den konvertierten Wert über die Adapterverbindung und zeigt ihn auf dem Terminal (z.B. Visualisierung) unter der Objekt-ID OutputNumber_N1 an.

Programmablauf und Verbindungen

Die Verdrahtung der Bausteine erfolgt über Adapterverbindungen. Initial wird der Preset-Wert 10 über AI_INT_TO_I bereitgestellt, sobald der Load-Eingang (Input_LD) ein INITO-Ereignis auslöst. Der Zähler startet mit PV=10.

Ablauf:

1. **Zählen**: Eine positive Flanke am Eingang I1 (Input_CD) sendet ein Ereignis über die Adapterverbindung an den CD-Eingang von AI_FB_CTD. Der Zähler dekrementiert um 1.
2. **Laden**: Eine positive Flanke am Eingang I2 (Input_LD) löst das LD-Ereignis aus und setzt den Zähler zurück auf den Wert von PV (10). Gleichzeitig wird über INITO der Konverter AI_INT_TO_I getriggert, um den PV-Wert erneut zu setzen.
3. **Ausgabe**: Der aktuelle Zählerwert (CV) wird über AI_TO_AUDI in ein Terminal-Format gewandelt und auf der Visualisierung (Q_NumericValue_AUDI) ausgegeben. Der Ausgang Q des Zählers wird auf den digitalen Ausgang Q1 geschaltet.

Hinweise aus dem Quelltext:

- Der Baustein AI_TO_AUDI kann keine negativen Werte verarbeiten. Da ein Rückwärtszähler unter Null zählen kann, ist dies eine Einschränkung.
- Es wird empfohlen, einen AX_D_FF (Ereignis-Flipflop) einzubauen, um die Anzahl der Ereignisse am Terminal zu reduzieren.

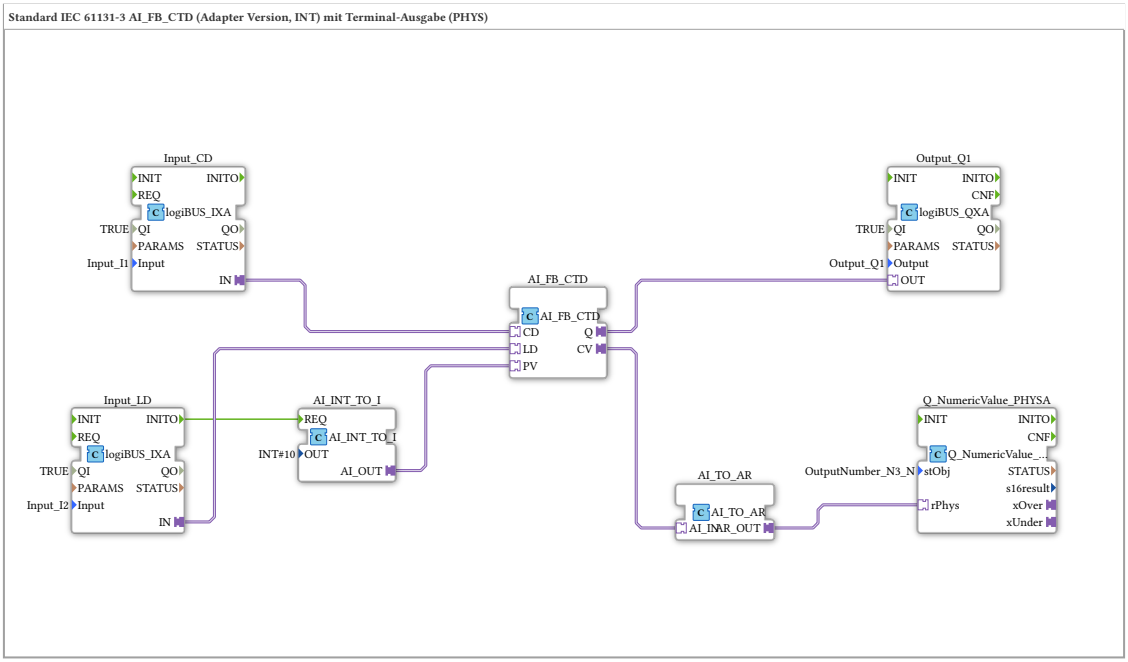
Zusammenfassung

Diese Übung vermittelt den Umgang mit einem IEC 61131-3 Rückwärtszähler (CTD) in einer Adapter-basierten Implementierung unter 4diac. Sie zeigt die Einbindung digitaler Ein- und Ausgänge über logiBUS sowie die Visualisierung von Zählerwerten auf einem Terminal. Der Lernende versteht das Zusammenspiel von Ereignissen, Datenkonvertierung und die Grenzen der verwendeten Bausteine (keine negativen Zahlen). Die Übung eignet sich für Fortgeschrittene mit Grundkenntnissen in 4diac und logiBUS.

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

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

2.1.2.499 Uebung_215b_AR: Standard IEC 61131-3 AI_FB_CTD (Adapter Version, INT) mit Terminal-Ausgabe (PHYS)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung implementiert einen Abwärtszähler (CTD) nach IEC 61131-3 als Adaptervariante. Der Zähler wird über zwei digitale Eingänge (Count-Down und Load) gesteuert und gibt einen digitalen Ausgang (Q) sowie den aktuellen Zählerwert (CV) aus. Der Zählerwert wird über einen Konverter in eine Textdarstellung umgewandelt und auf einem Terminal (PHYS) ausgegeben. Der Preset-Wert (PV) wird fest auf 10 gesetzt und dem Zähler über einen weiteren Adapter zugeführt.

Lernziele:

- Verständnis des Funktionsbausteins AI_FB_CTD (Abwärtszähler als Adapter)
- Umgang mit Adapter-Schnittstellen zur Daten- und Ereignisübertragung
- Datentypkonvertierung (INT nach Adapter, CV nach Array)
- Steuerung von logiBUS-Ein- und Ausgängen
- Ausgabe numerischer Werte auf einem Terminal

Schwierigkeitsgrad: Fortgeschritten Vorkenntnisse: Grundlagen der IEC 61131-3, 4diac-IDE, Adapterkonzept

Verwendete Funktionsbausteine (FBs)

AI_FB_CTD

- **Typ**: `adapter::iec61131::counters::AI_FB_CTD`
- **Verwendete interne FBs**: Keine (Basisfunktionsbaustein)
- **Funktionsweise**: Abwärtszähler (Count Down) mit den Adapter-Schnittstellen `CD` (Count-Eingang), `LD` (Preset laden), `PV` (Presetwert), `Q` (Ausgang, wird TRUE wenn CV=0) und `CV` (aktueller Zählerwert).

AI_INT_TO_I

- **Typ**: `adapter::conversion::unidirectional::AI_INT_TO_I`
- **Parameter**:
  - `OUT` = `INT#10` (Presetwert, fest auf 10 gesetzt)
- **Funktionsweise**: Wandelt einen Integer-Wert (hier 10) in einen Adapterausgang um, der mit dem PV-Eingang des Zählers verbunden wird.

Input_CD

- **Typ**: `logiBUS::io::DI::logiBUS_IXA`
- **Parameter**:
  - `QI` = `TRUE` (Freigabe)
  - `Input` = `Input_I1` (physischer Eingang 1)
- **Funktionsweise**: Digitaler Eingang für das CD-Signal (Count Down). Bei positiver Flanke wird der Zähler dekrementiert.

Input_LD

- **Typ**: `logiBUS::io::DI::logiBUS_IXA`
- **Parameter**:
  - `QI` = `TRUE`
  - `Input` = `Input_I2` (physischer Eingang 2)
- **Funktionsweise**: Digitaler Eingang für das LD-Signal (Load). Bei positiver Flanke wird der Presetwert (PV) in den Zähler geladen.

Output_Q1

- **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
- **Parameter**:
  - `QI` = `TRUE`
  - `Output` = `Output_Q1` (physischer Ausgang 1)
- **Funktionsweise**: Digitaler Ausgang, der den Q-Zustand des Zählers (TRUE=CV=0) an ein logiBUS-Ausgabemodul weiterleitet.

AI_TO_AR

- **Typ**: `adapter::conversion::unidirectional::AI_TO_AR`
- **Funktionsweise**: Wandelt den analogen Adapterwert (CV) des Zählers in ein Array (AR) um, das vom Terminalbaustein erwartet wird.

Q_NumericValue_PHYSA

- **Typ**: `isobus::UT::Q::Q_NumericValue_PHYSA`
- **Parameter**:
  - `stObj` = `OutputNumber_N3` (referenziert ein Terminal-Ausgabeobjekt)
- **Funktionsweise**: Zeigt den übergebenen numerischen Wert (aus AI_TO_AR) auf dem Terminal (PHYS) an.

Programmablauf und Verbindungen

Der Programmablauf besteht aus einem ereignisgesteuerten und einem datenflussgesteuerten Teil.

1. **Initialisierung**: Beim Start wird über den Eventausgang `INIT0` von `Input_LD` der Konverter `AI_INT_TO_I` getriggert (`REQ`). Dieser konvertiert den festen Integerwert `10` in einen Adapterwert und übergibt ihn an den `PV`-Eingang von `AI_FB_CTD`.
2. **Zählerbetrieb**:
   - Ein steigendes Signal an `Input_I1` (CD) wird über `Input_CD` an den `CD`-Adaptereingang des Zählers weitergeleitet. Der Zähler dekrementiert den aktuellen Wert.
   - Ein steigendes Signal an `Input_I2` (LD) triggert erneut `AI_INT_TO_I` und lädt den Presetwert in den Zähler.
   - Der Ausgang `Q` des Zählers wird direkt mit `Output_Q1` verbunden: Sobald der Zählerstand 0 erreicht, wird der Ausgang gesetzt.
   - Der aktuelle Zählerwert `CV` wird über `AI_TO_AR` in ein Array umgewandelt und an den Terminalbaustein übergeben, der den Zahlenwert auf dem Bildschirm ausgibt.
3. **Hinweise aus den Kommentaren**:
   - Bei dieser Konfiguration können auch negative Werte auftreten (CD-Signale bei CV=0).
   - Gegebenenfalls sollte ein `AX_D_FF` (Flankenmerker) zwischengeschaltet werden, um die Ereignisse zu reduzieren, falls der Zähler zu schnell zählt.

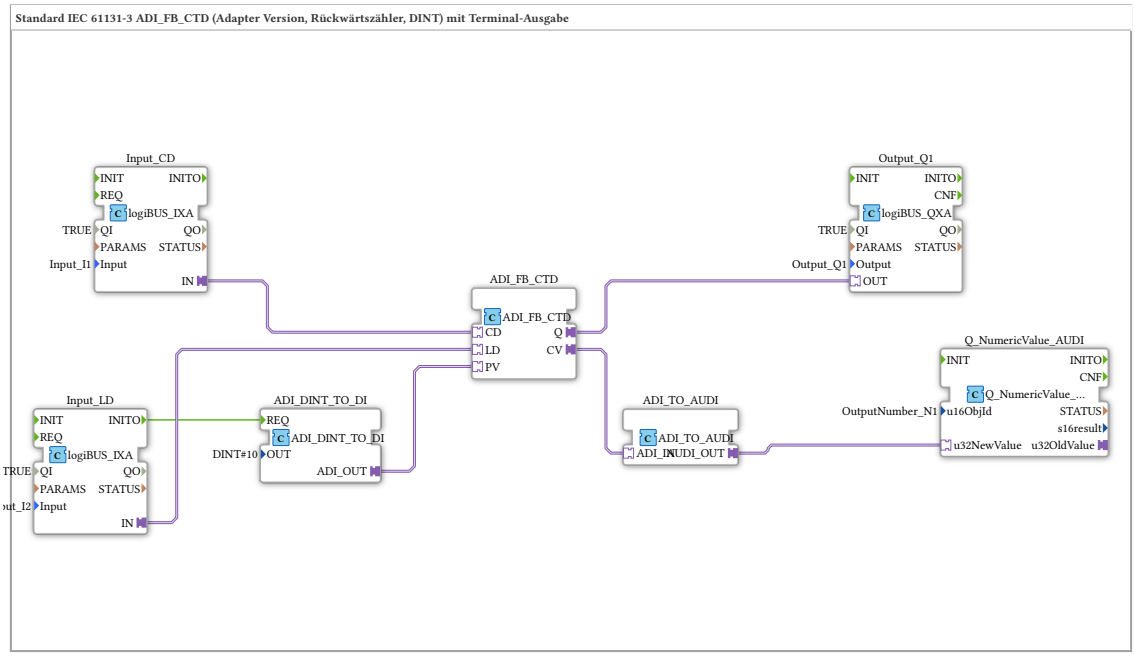
Zusammenfassung

Die Übung zeigt die Realisierung eines Abwärtszählers nach IEC 61131-3 unter Verwendung von Adaptern zur Verbindung von logiBUS-Ein-/Ausgängen, Datentypkonvertierung und Terminalausgabe. Der Zähler wird über zwei digitale Eingänge gesteuert und gibt sowohl einen binären Ausgang als auch einen numerischen Wert aus. Die Lösung verdeutlicht den modularen Aufbau mit wiederverwendbaren Adapterbausteinen und die einfache Integration von Hardware-Schnittstellen in die 4diac-IDE.

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

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

2.1.2.500 Uebung_216_ADI: Standard IEC 61131-3 ADI_FB_CTD (Adapter Version, Rückwärtszähler, DINT) mit Terminal-Ausgabe



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung realisiert einen Rückwärtszähler (Count Down) nach IEC 61131-3 auf Basis des Adapter-Funktionsbausteins `ADI_FB_CTD`. Der Zähler arbeitet mit dem Datentyp `DINT` und gibt den aktuellen Zählerstand sowie das Zählerende-Signal aus. Zur Visualisierung des Zählerstands wird der Wert über einen Terminal-Baustein ausgegeben, wobei eine Adapter-Konvertierung notwendig ist.

Die Übung zeigt den Einsatz von Adapter-Schnittstellen zur Verbindung verschiedener Funktionsbausteine sowie die Grenzen der verwendeten Konvertierung.

Verwendete Funktionsbausteine (FBs)

ADI_FB_CTD

- **Typ**: `adapter::iec61131::counters::ADI_FB_CTD`
- **Parameter**: keine
- **Funktion**: IEC 61131-3 Rückwärtszähler (CTD). Zählt bei jedem positiven Flanke am Eingang `CD` den aktuellen Wert (PV) herunter. Bei Erreichen von Null wird der Ausgang `Q` gesetzt. Über den Eingang `LD` wird der Zähler mit dem Wert von `PV` geladen.
- **Adapter-Eingänge**: `CD` (Zählimpuls), `LD` (Laden), `PV` (Vorgabewert)
- **Adapter-Ausgänge**: `Q` (Zählerende), `CV` (aktueller Zählerstand)

ADI_DINT_TO_DI

- **Typ**: `adapter::conversion::unidirectional::ADI_DINT_TO_DI`

- **Parameter**: `OUT = DINT#10`
- **Funktion**: Wandelt eine konstante DINT-Zahl (hier 10) in ein DI-Adapter-Interface um, das als Vorgabewert (`PV`) für den Zähler dient.

Input_CD
- **Typ**: `logiBUS::io::DI::logiBUS_IXA`
- **Parameter**: `QI = TRUE`, `Input = Input_I1`
- **Funktion**: Digitaler Eingang für das Zählsignal (CD). Aktiviert über den physischen Eingang I1.

Input_LD
- **Typ**: `logiBUS::io::DI::logiBUS_IXA`
- **Parameter**: `QI = TRUE`, `Input = Input_I2`
- **Funktion**: Digitaler Eingang für das Ladesignal (LD). Aktiviert über den physischen Eingang I2. Der Ereignisausgang `INITO` dieses Bausteins startet die Initialisierung des Vorgabewerts.

Output_Q1
- **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
- **Parameter**: `QI = TRUE`, `Output = Output_Q1`
- **Funktion**: Digitaler Ausgang für das Zählerende-Signal (`Q`). Schaltet den physischen Ausgang Q1.

ADI_TO_AUDI
- **Typ**: `adapter::conversion::unidirectional::ADI_TO_AUDI`
- **Parameter**: keine
- **Funktion**: Konvertiert das ADI-Interface (DINT) in ein AUDI-Interface (Analog Universal Data Interface). **Wichtig:** Diese Konvertierung unterstützt keine negativen Zahlen – der Zählerstand kann nur als positive Zahl oder Null dargestellt werden.

Q_NumericValue_AUDI
- **Typ**: `isobus::UT::Q::Q_NumericValue_AUDI`
- **Parameter**: `u16ObjId = OutputNumber_N1`
- **Funktion**: Gibt den übergebenen numerischen Wert (vom AUDI-Interface) auf einem Terminal aus. Die Objekt-ID verweist auf eine vordefinierte Ausgabeadresse.

Programmablauf und Verbindungen

Die Steuerung erfolgt über Ereignis- und Adapterverbindungen:

1. **Initialisierung des Vorgabewerts**: Der Ereignisausgang `INITO` des Bausteins `Input_LD` triggert den Baustein `ADI_DINT_TO_DI`. Dadurch wird die Konstante `DINT#10` einmalig in den `ADI_FB_CTD` geladen (über die Adapterverbindung `ADI_DINT_TO_DI.ADI_OUT` → `ADI_FB_CTD.PV`).
2. **Zählimpulse (CD)**: Der Adapterausgang `Input_CD.IN` ist mit dem Adaptereingang `ADI_FB_CTD.CD` verbunden. Jede positive Flanke am Eingang I1 zählt den Zählerstand um eins herunter.
3. **Ladesignal (LD)**: Der Adapterausgang `Input_LD.IN` ist mit dem Adaptereingang `ADI_FB_CTD.LD` verbunden. Ein Signal an I2 lädt den Zähler mit dem aktuellen Vorgabewert (10).
4. **Zählerende (Q)**: Der Adapterausgang `ADI_FB_CTD.Q` führt zum Adaptereingang `Output_Q1.OUT`. Bei Erreichen von Null wird der Ausgang Q1 aktiviert.
5. **Ausgabe des Zählerstands**: Der Adapterausgang `ADI_FB_CTD.CV` (aktueller Zählerstand) wird über `ADI_TO_AUDI` konvertiert und an den Terminal-Baustein `Q_NumericValue_AUDI` weitergeleitet. Der Zählerstand erscheint auf dem Terminal.

Hinweis: Der Kommentar im Netzwerk weist darauf hin, dass der Baustein `ADI_TO_AUDI` keine negativen Zahlen verarbeiten kann. Da der Rückwärtszähler nur bis Null zählt, tritt dieser Fall in der Übung nicht auf. Für erweiterte Anwendungen müsste eine geeignetere Konvertierung gewählt werden.

Zusammenfassung

Die Übung vermittelt den Umgang mit dem IEC 61131-3-Rückwärtszähler als Adapter-Baustein. Sie zeigt:

- die Verwendung von Adapter-Schnittstellen zur Verbindung von Digital-Eingängen, Zähler und Ausgängen,
- die Initialisierung eines Vorgabewerts über eine Konvertierung,

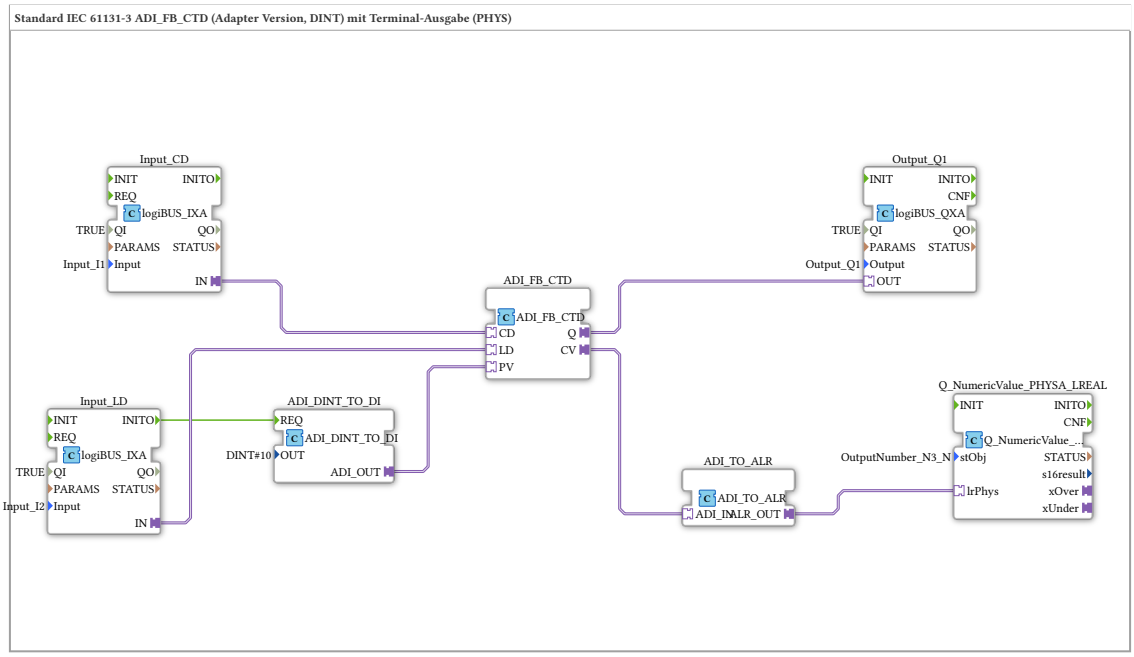
- die Terminal-Ausgabe eines Zählerstands mit Hilfe eines AUDI-Interfaces,
- die Einschränkungen der verwendeten Konvertierung (`ADI_TO_AUDI`) bei negativen Werten.

Der Aufbau ist als Subapplikation realisiert und kann direkt in eine 4diac-IDE-Umgebung geladen und getestet werden.

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

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

2.1.2.501 Uebung_216b_ALR: Standard IEC 61131-3 ADI_FB_CTD (Adapter Version, DINT) mit Terminal-Ausgabe (PHYS)



• – ○ • – ○ • – ○ •

Einleitung

Die Übung **Uebung_216b_ALR** realisiert einen Abwärtszähler (Counter Down) gemäß IEC 61131-3 unter Verwendung eines Adapter-basierten Funktionsbausteins `ADI_FB_CTD`. Der Zählerwert wird über eine Adapter-Wandlungskette auf ein alphanumerisches Terminal (PHYS) ausgegeben. Zusätzlich wird ein digitaler Ausgang gesetzt, wenn der Zählerstand Null erreicht. Die Übung veranschaulicht die Integration von logiBUS-Eingängen, Adapter-Konvertierungen und Terminal-Ausgabe in einer kompakten Subapplikation.

Verwendete Funktionsbausteine (FBs)

Interne Funktionsbausteine

- **ADI_FB_CTD** (Typ: `adapter::iec61131::counters::ADI_FB_CTD`)
  - **Beschreibung**: Adapter-Version eines IEC 61131-3 Abwärtszählers (CTD). Der Baustein zählt bei jedem fallenden Flanke am Ereigniseingang `CD` den aktuellen Wert (`CV`) um 1 herunter. Der Preset-Wert wird über den Adapter-Eingang `PV` geladen, sobald der Eingang `LD` aktiviert wird. Der Ausgang `Q` wird gesetzt, sobald `CV` den Wert 0 erreicht.
  - **Parameter**: (keine expliziten Parameter gesetzt – verwendet Standardwerte)
  - **Ereignis-/Datenschnittstellen**:
    - Ereigniseingang: `CD` (Count Down), `LD` (Load)
    - Adapter-Dateneingang: `PV` (Preset Value, DINT)
    - Adapter-Datenausgang: `CV` (Current Value, DINT), `Q` (BOOL)
- **ADI_DINT_TO_DI** (Typ: `adapter::conversion::unidirectional::ADI_DINT_TO_DI`)

  - **Beschreibung**: Wandelt einen DINT-Wert in einen Adapter-Dateneingang (DI) um. Hier wird ein fester Preset-Wert von DINT#10 bereitgestellt.
  - **Parameter**: `OUT` = DINT#10
  - **Ereignis-/Datenschnittstellen**:
    - Ereigniseingang: `REQ` (Trigger zur Ausgabe)
    - Adapter-Datenausgang: `ADI_OUT` (DINT)
- **Input_CD** (Typ: `logiBUS::io::DI::logiBUS_IXA`)
  - **Beschreibung**: Digitaler Eingangsbaustein für logiBUS, der das Signal des physischen Eingangs `Input_I1` bereitstellt.
  - **Parameter**: `QI` = TRUE (Qualifier), `Input` = Input_I1
  - **Adapter-Ausgang**: `IN` (digitale Information)
- **Input_LD** (Typ: `logiBUS::io::DI::logiBUS_IXA`)
  - **Beschreibung**: Digitaler Eingangsbaustein für logiBUS, der das Signal des physischen Eingangs `Input_I2` bereitstellt.
  - **Parameter**: `QI` = TRUE, `Input` = Input_I2
  - **Adapter-Ausgang**: `IN`
  - **Ereignisausgang**: `INITO` (wird bei Initialisierung ausgelöst)
- **Output_Q1** (Typ: `logiBUS::io::DQ::logiBUS_QXA`)
  - **Beschreibung**: Digitaler Ausgangsbaustein für logiBUS, der den physischen Ausgang `Output_Q1` ansteuert.
  - **Parameter**: `QI` = TRUE, `Output` = Output_Q1
  - **Adapter-Eingang**: `OUT` (digitale Information)
- **ADI_TO_ALR** (Typ: `adapter::conversion::unidirectional::ADI_TO_ALR`)
  - **Beschreibung**: Wandelt einen Adapter-Datenwert (DINT) in ein alphanumerisches Format (ALR) um, das für die Terminalausgabe geeignet ist.
  - **Keine Parameter** gesetzt.
  - **Schnittstellen**:
    - Adapter-Eingang: `ADI_IN` (DINT)
    - Adapter-Ausgang: `ALR_OUT` (ALR)
- **Q_NumericValue_PHYSA_LREAL** (Typ: `isobus::UT::Q::Q_NumericValue_PHYSA_LREAL`)
  - **Beschreibung**: Gibt einen numerischen Wert (als LREAL interpretiert) auf einem physischen Terminal aus. Der Wert wird vom angeschlossenen `stObj` (hier `OutputNumber_N3`) dargestellt.
  - **Parameter**: `stObj` = OutputNumber_N3 (Referenz auf ein Terminal-Objekt)
  - **Adapter-Eingang**: `lrPhys` (physikalischer Wert als LREAL)

Programmablauf und Verbindungen

1. **Initialisierung**: Beim Start der Subapplikation wird der Baustein `Input_LD` aktiv und löst das Ereignis `INITO` aus. Dieses Ereignis triggert `ADI_DINT_TO_DI.REQ`, sodass der Preset-Wert (DINT#10) am Adapterausgang `ADI_OUT` anliegt.
2. **Preset laden**: Der Preset-Wert wird über die Adapterverbindung an den Eingang `PV` des Zählers `ADI_FB_CTD` übergeben. Gleichzeitig wird durch das Ereignis `INITO` der Lade-Eingang `LD` des Zählers aktiviert? (Die Ereignisverbindung `Input_LD.INITO` geht nur an `ADI_DINT_TO_DI.REQ`, nicht direkt an den Zähler. Allerdings ist `Input_LD.IN` mit `ADI_FB_CTD.LD` verbunden – diese Verbindung ist als Adapterverbindung ausgeführt und überträgt das digitale Signal. Die Initialisierung von `Input_LD` setzt vermutlich den Eingang `LD` auf TRUE, sodass der Zähler den Preset lädt.)
3. **Zählbetrieb**: Der digitale Eingang `Input_CD` (Pin I1) führt dem Zähler über den Adaptereingang `CD` die Zählimpulse zu. Bei jeder fallenden Flanke (bzw. gemäß Definition des Adapters) verringert der Zähler den aktuellen Wert `CV` um 1.
4. **Ausgangsstatus**: Sobald `CV` auf 0 gefallen ist, setzt der Zähler den Ausgang `Q` auf TRUE. Dieser wird über `Output_Q1` an den physischen Ausgang Q1 weitergegeben.
5. **Terminalausgabe**: Der aktuelle Zählerstand (`CV`) wird über den Adapter `ADI_TO_ALR` in ein alphanumerisches Format gewandelt und an den Terminalbaustein `Q_NumericValue_PHYSA_LREAL` übergeben. Dieser gibt den Wert auf dem konfigurierten Terminal `OutputNumber_N3` aus. Beachte den Kommentar: Hier sind auch negative Werte möglich (durch Überlauf des Zählers unter 0).

6. **Hinweise**: Der Kommentar schlägt vor, ggf. einen `AX_D_FF` (FlipFlop) einzubauen, um die Ereignisrate zu reduzieren. Dies wäre bei sehr schnellen Zählimpulsen sinnvoll, um die Terminalausgabe zu entlasten.

Verbindungsübersicht (Adapterverbindungen)

Von (Quelle)	Nach (Ziel)	Bemerkung
`Input_CD.IN`	`ADI_FB_CTD.CD`	Zählimpulse
`Input_LD.IN`	`ADI_FB_CTD.LD`	Ladesignal
`ADI_FB_CTD.Q`	`Output_Q1.OUT`	Ausgangsstatus (CV=0)
`ADI_FB_CTD.CV`	`ADI_TO_ALR.ADI_IN`	aktueller Zählerstand
`ADI_TO_ALR.ALR_OUT`	`Q_NumericValue_PHYSA_LREAL.lrPhys`	Terminalausgabe
`ADI_DINT_TO_DI.ADI_OUT`	`ADI_FB_CTD.PV`	Preset-Wert

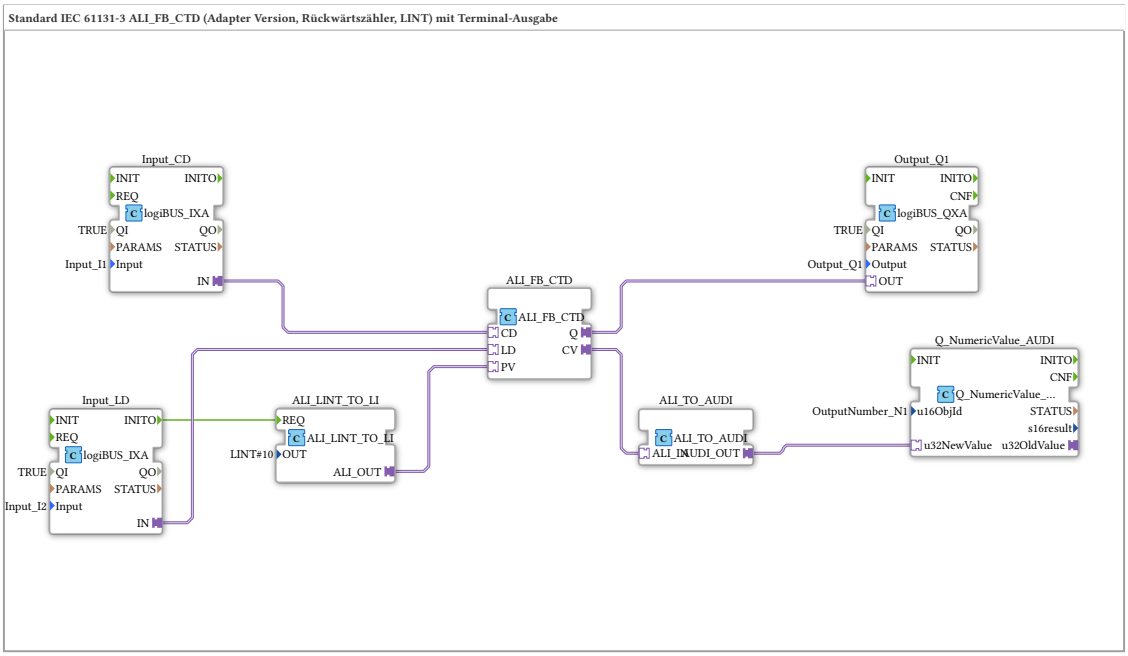
Ereignisverbindung:

- `Input_LD.INITO` ⟶ `ADI_DINT_TO_DI.REQ`

Zusammenfassung

Die Übung demonstriert den Einsatz eines IEC-61131-3-Abwärtszählers in einer Adapter-basierten Umgebung. Durch die Kombination von logiBUS-Ein-/Ausgängen, einer DINT-Konvertierung und einer alphanumerischen Terminalausgabe wird ein vollständiger, praxisnaher Zählerkreislauf abgebildet. Der Benutzer lernt, Adapterverbindungen zu verschalten und Ereignissteuerungen zu nutzen. Ein besonderer Fokus liegt auf der korrekten Initialisierung des Preset-Werts und der Ausgabe des Zählerstands (auch negativer Werte) auf einem Terminal.

2.1.2.502 Uebung_217_ALI: Standard IEC 61131-3 ALI_FB_CTD (Adapter Version, Rückwärtszähler, LINT) mit Terminal-Ausgabe



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung realisiert einen **IEC 61131-3 konformen Rückwärtszähler (CTD) in Adapterversion** für den Datentyp LINT (Long Integer). Der aktuelle Zählerstand wird auf einem Terminal ausgegeben. Der Zähler wird über ein digitales Eingangssignal **CD** (Count Down) dekrementiert. Ein weiteres digitales

Signal **LD** (Load) lädt den Zähler mit einem vorgegebenen Preset-Wert. Sobald der Zählerstand den Wert 0 erreicht, wird der Ausgang **Q** gesetzt.

Der Preset-Wert wird bei der Initialisierung aus einem Konstanten-Baustein (LINT#10) bezogen und über einen ALI-LINT-to-LI-Konverter an den Zähler übergeben. Der Zählerstand wird über einen ALI-to-AUDI-Konverter an einen numerischen Anzeigebaustein (Q_NumericValue_AUDI) weitergeleitet und auf dem Terminal ausgegeben.

Hinweis: Der Konverter ALI_TO_AUDI unterstützt keine negativen Zahlen – dies ist bei der Wahl der Preset-Werte zu beachten.

Verwendete Funktionsbausteine (FBs)

ALI_FB_CTD

- **Typ:** `adapter::iec61131::counters::ALI_FB_CTD`
- **Verwendete interne FBs:** keine (eigenständiger Zählerbaustein)
- **Parameter:**
  - Ereigniseingänge: CD, LD, R (nicht genutzt)
  - Ereignisausgänge: Q
  - Dateneingänge:
    - CD (Adapter) – Zählimpuls (Count Down)
    - LD (Adapter) – Ladesignal (Load)
    - PV (Adapter: LINT) – Preset-Wert (Vorgabewert)
  - Datenausgänge:
    - CV (Adapter: LINT) – aktueller Zählerstand
    - Q (Adapter: BOOL) – Ausgang wird TRUE, wenn CV = 0
- **Funktionsweise:** Ein Impuls am Ereigniseingang CD dekrementiert den aktuellen Zählerstand CV um 1. Ein Impuls am LD lädt den Wert von PV in CV. Wenn CV = 0 wird der Ausgang Q gesetzt.

ALI_LINT_TO_LI

- **Typ:** `adapter::conversion::unidirectional::ALI_LINT_TO_LI`
- **Verwendete interne FBs:** keine
- **Parameter:**
  - Parameter: `OUT` = `LINT#10` (Ausgangswert wird als Konstante bereitgestellt)
  - Ereigniseingang: REQ
  - Ereignisausgang: CNF
  - Dateneingang: IN (LINT) – nicht verbunden, da über Parameter konstant
  - Datenausgang: ALI_OUT (Adapter) – liefert den LINT-Wert
- **Funktionsweise:** Stellt bei einer Anforderung (REQ) den parametrierten LINT-Wert (hier 10) am Ausgang ALI_OUT bereit. Dient als Quelle für den Preset-Wert PV.

Input_CD

- **Typ:** `logiBUS::io::DI::logiBUS_IXA`
- **Verwendete interne FBs:** keine
- **Parameter:**
  - `QI` = `TRUE` (Eingang aktiviert)
  - `Input` = `Input_I1` (Benennung des physikalischen Eingangs)
- **Funktionsweise:** Wandelt ein digitales Signal vom Feldbus (Input_I1) in einen Adapter-Ereignisausgang IN. Wird verwendet, um das CD-Signal (Count Down) an den Zähler zu liefern.

Input_LD

- **Typ:** `logiBUS::io::DI::logiBUS_IXA`
- **Parameter:**
  - `QI` = `TRUE`
  - `Input` = `Input_I2`
- **Funktionsweise:** Analog zu Input_CD, liefert das Ladesignal (LD) für den Zähler.

Output_Q1

- **Typ:** `logiBUS::io::DQ::logiBUS_QXA`

- **Parameter:**
  - `QI` = `TRUE`
  - `Output` = `Output_Q1`
- **Funktionsweise:** Nimmt den digitalen Ausgang Q des Zählers entgegen und stellt ihn als Feldbus-Ausgang (Output_Q1) bereit.

ALI_TO_AUDI

- **Typ:** `adapter::conversion::unidirectional::ALI_TO_AUDI`
- **Verwendete interne FBs:** keine
- **Parameter:** keine
- **Funktionsweise:** Wandelt den ALI-Adatper (LINT) in einen AUDI-Adapter (UINT?) um. Der konvertierte Wert wird an den numerischen Anzeigebaustein weitergegeben. Da der Konverter nur positive Werte verarbeitet, können negative Zählerstände nicht angezeigt werden.

Q_NumericValue_AUDI

- **Typ:** `isobus::UT::Q::Q_NumericValue_AUDI`
- **Parameter:**
  - `u16ObjId` = `OutputNumber_N1` (Kennung des Terminal-Ausgabeelements)
- **Funktionsweise:** Nimmt einen AUDI-Adapter mit einem numerischen Wert entgegen und zeigt diesen auf dem angeschlossenen Terminal unter der Objekt-ID OutputNumber_N1 an.

Programmablauf und Verbindungen

Die folgende Abbildung zeigt den logischen Daten- und Ereignisfluss:

- **Initialisierung:** Beim Start (Ereignis INITO von Input_LD) wird der Baustein ALI_LINT_TO_LI über den Ereigniseingang REQ getriggert. Dieser liefert am Ausgang ALI_OUT den konstanten Wert LINT#10. Dieser ist mit dem PV-Eingang des Zählers verbunden.
- **Laden des Preset-Werts:** Ein Impuls an Input_LD (Eingang Input_I2) löst den Ereignisausgang INITO und gleichzeitig über den Adapter IN das Ladesignal LD des Zählers aus. Der Zähler übernimmt den Wert von PV und setzt CV = 10.
- **Count Down:** Ein Impuls an Input_CD (Eingang Input_I1) wirkt auf den Adapter CD des Zählers. Jeder Impuls verringert CV um 1.
- **Ausgabe Zählerstand:** Der aktuelle Zählerstand CV wird über den Adapter CV an ALI_TO_AUDI weitergeleitet. Dieser wandelt den Wert in den AUDI-Adapter um und übergibt ihn an Q_NumericValue_AUDI, der ihn auf dem Terminal anzeigt.
- **Ausgang Q:** Wenn CV = 0 erreicht, setzt der Zähler den Ausgang Q. Dieser wird über Output_Q1 als Feldbus-Signal (Output_Q1) ausgegeben.

Verbindungsliste (Adapter- und Ereignisverbindungen)

- **Ereignisverbindung:** `Input_LD.INITO` ⟶ `ALI_LINT_TO_LI.REQ`
- **Adapterverbindungen:**
  - `Input_CD.IN` ⟶ `ALI_FB_CTD.CD`
  - `Input_LD.IN` ⟶ `ALI_FB_CTD.LD`
  - `ALI_FB_CTD.Q` ⟶ `Output_Q1.OUT`
  - `ALI_FB_CTD.CV` ⟶ `ALI_TO_AUDI.ALI_IN`
  - `ALI_TO_AUDI.AUDI_OUT` ⟶ `Q_NumericValue_AUDI.u32NewValue`
  - `ALI_LINT_TO_LI.ALI_OUT` ⟶ `ALI_FB_CTD.PV`

Zusammenfassung

In dieser Übung wird ein klassischer Rückwärtszähler (CTD) nach IEC 61131-3 in Adapterbauweise umgesetzt. Die Teilnehmer lernen:

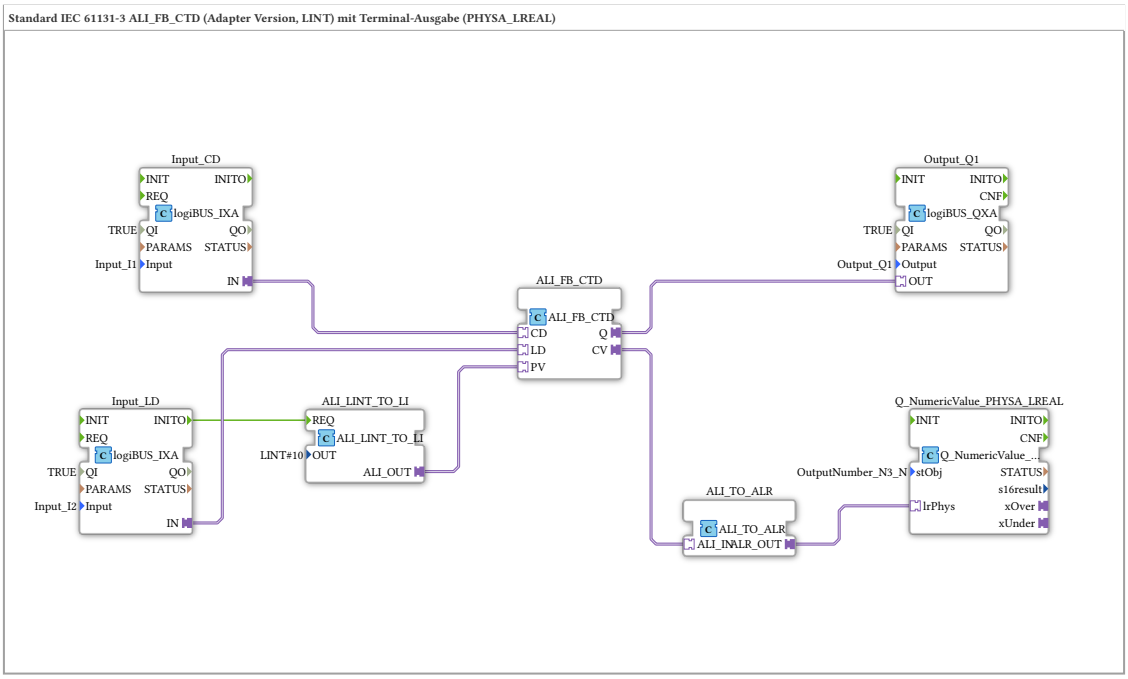
- Die Verwendung eines Zähler-FB mit den Ereignissen CD und LD.
- Das Initialisieren eines Preset-Wertes über einen separaten Konvertierungsbaustein (ALI_LINT_TO_LI).
- Die Weiterleitung von Zählerständen an eine Terminalausgabe (Q_NumericValue_AUDI) unter Berücksichtigung der Einschränkungen des ALI_TO_AUDI-Konverters (keine negativen Zahlen).
- Die Verschaltung von Adaptern und Ereignissen in einem SubApp-Netzwerk.

Der Schwierigkeitsgrad ist **mittel**. Vorkenntnisse in der 4diac-IDE und grundlegendes Verständnis von Ereignis-/Datenflüssen sowie Adaptern werden empfohlen. Die Übung kann direkt nach dem Import der Bibliothek und dem Anlegen eines passenden Systems gestartet werden.

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

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

2.1.2.503 Uebung_217b_ALR: Standard IEC 61131-3 ALI_FB_CTD (Adapter Version, LINT) mit Terminal-Ausgabe (PHYSA_LREAL)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert die Verwendung eines IEC 61131-3 konformen Abwärtszählers (CTD) im Adapter-Format (ALI) mit dem Datentyp LINT. Der Zählerwert wird über einen Eingangstaster dekrementiert, ein weiterer Taster lädt einen vorgegebenen Preset-Wert. Der aktuelle Zählerstand wird über eine Umwandlungskette (LINT ⟶ LREAL) auf einem Terminal ausgegeben. Der Ausgang des Zählers (Q) schaltet einen digitalen Ausgang.

Die Übung veranschaulicht den Umgang mit Adapter-Schnittstellen, die Signalumwandlung zwischen unterschiedlichen Datentypen sowie die Anbindung einer numerischen Terminalausgabe.

Verwendete Funktionsbausteine (FBs)

ALI_FB_CTD

- **Typ**: `adapter::iec61131::counters::ALI_FB_CTD`
- **Beschreibung**: IEC 61131-3 Abwärtszähler (CTD) für den Datentyp LINT. Er besitzt zwei Ereigniseingänge (`CD`, `LD`), einen Dateneingang für den Preset-Wert (`PV`) und einen Ereignisausgang (`Q`) für das Zählergebnis. Der Zählerstand wird am Adapterausgang `CV` bereitgestellt.

ALI_LINT_TO_LI

- **Typ**: `adapter::conversion::unidirectional::ALI_LINT_TO_LI`
- **Beschreibung**: Konvertiert einen LINT-Wert in einen LINT-Adapterwert (LI). Wird verwendet, um den Preset-Wert (LINT#10) an den Zähler zu übergeben.
- **Parameter**: `OUT` = `LINT#10` (voreingestellter Zählerendwert)

Input_CD (logiBUS_IXA)

- **Typ**: `logiBUS::io::DI::logiBUS_IXA`
- **Beschreibung**: Digitaler Eingangsbaustein für den logiBUS. Erfasst den Zustand des Tasters `Input_I1`, der das Dekrementieren (`CD`) auslöst.

- **Parameter**: `QI` = `TRUE`, `Input` = `Input_I1`

Input_LD (logiBUS_IXA)

- **Typ**: `logiBUS::io::DI::logiBUS_IXA`
- **Beschreibung**: Zweiter digitaler Eingang für den Taster `Input_I2`, der das Laden des Preset-Wertes (`LD`) auslöst.
- **Parameter**: `QI` = `TRUE`, `Input` = `Input_I2`

Output_Q1 (logiBUS_QXA)

- **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
- **Beschreibung**: Digitaler Ausgangsbaustein für den logiBUS. Gibt den Zustand von `Q` (Zählerstand ≤ 0) auf `Output_Q1` aus.
- **Parameter**: `QI` = `TRUE`, `Output` = `Output_Q1`

ALI_TO_ALR

- **Typ**: `adapter::conversion::unidirectional::ALI_TO_ALR`
- **Beschreibung**: Wandelt einen ALI-Adapter (ganzzahlig) in einen ALR-Adapter (reell) um. Ermöglicht die Übergabe des LINT-Zählerwerts an eine Terminalausgabe, die einen LREAL-Wert erwartet.

Q_NumericValue_PHYSA_LREAL

- **Typ**: `isobus::UT::Q::Q_NumericValue_PHYSA_LREAL`
- **Beschreibung**: Terminalausgabe-Baustein für numerische Werte vom Typ `PHYSA_LREAL`. Zeigt den umgewandelten Zählerwert auf dem zugeordneten Terminalobjekt `OutputNumber_N3` an.
- **Parameter**: `stObj` = `OutputNumber_N3`

Programmablauf und Verbindungen

1. **Initialisierung**: Beim Start (z. B. nach einem Reset) sendet `Input_LD` das Ereignis `INITO` an `ALI_LINT_TO_LI`, welches den Preset-Wert `LINT#10` bereitstellt.
2. **Preset laden**: Drücken des Tasters `Input_I2` erzeugt ein Ereignis auf dem Adapter `Input_LD.IN`. Dieses wird mit dem Ladeeingang `LD` des Zählers `ALI_FB_CTD` verbunden. Gleichzeitig wird der von `ALI_LINT_TO_LI` gelieferte Preset-Wert über den Adapter `ALI_OUT` an den Eingang `PV` des Zählers übergeben. Der Zähler übernimmt den Wert `10`.
3. **Dekrementieren**: Jeder Tastendruck auf `Input_I1` (Eingang `Input_CD`) erzeugt ein Ereignis, das über den Adapter `IN` an den Dekrementierungseingang `CD` des Zählers geleitet wird. Der Zählerstand verringert sich um 1.
4. **Ausgang Q**: Wenn der Zählerstand ≤ 0 ist, wird der Adapterausgang `Q` des Zählers aktiv. Dieses Signal wird an den digitalen Ausgang `Output_Q1` weitergegeben (Schalten einer Last oder Anzeige).
5. **Anzeige des Zählerstands**: Der aktuelle Zählerwert `CV` (Typ LINT) wird über die Umwandlungskette `ALI_TO_ALR` und `Q_NumericValue_PHYSA_LREAL` auf dem Terminal als Gleitkommazahl ausgegeben. Die Kommentare weisen darauf hin, dass hier auch negative Werte auftreten können und dass bei schnellen Ereignisfolgen ein AX_D_FF zur Reduzierung der Terminal-Aktualisierungen sinnvoll sein kann.

Verbindungsübersicht (Adapter- und Ereignisverbindungen):

Quelle	Ziel	Typ
`Input_CD.IN`	`ALI_FB_CTD.CD`	Adapter (IN)
`Input_LD.IN`	`ALI_FB_CTD.LD`	Adapter (IN)
`ALI_FB_CTD.Q`	`Output_Q1.OUT`	Adapter (OUT)
`ALI_FB_CTD.CV`	`ALI_TO_ALR.ALI_IN`	Adapter
`ALI_TO_ALR.ALR_OUT`	`Q_NumericValue_PHYSA_LREAL.lrPhys`	Adapter
`ALI_LINT_TO_LI.ALI_OUT`	`ALI_FB_CTD.PV`	Adapter
`Input_LD.INITO` (Ereignis)	`ALI_LINT_TO_LI.REQ`	Ereignis

Zusammenfassung

Die Übung **217b_ALR** vermittelt den sicheren Umgang mit einem IEC 61131-3 Abwärtszähler im Adapter-Format, der Signalumwandlung zwischen Ganzzahl- und Gleitkommatypen sowie der Anbindung einer Terminalausgabe. Sie zeigt die typische Verschaltung von logiBUS-Ein-/Ausgängen mit Funktionsbausteinen und bereitet auf die Realisierung zählerbasierter Steuerungsaufgaben vor.

Lernziele:

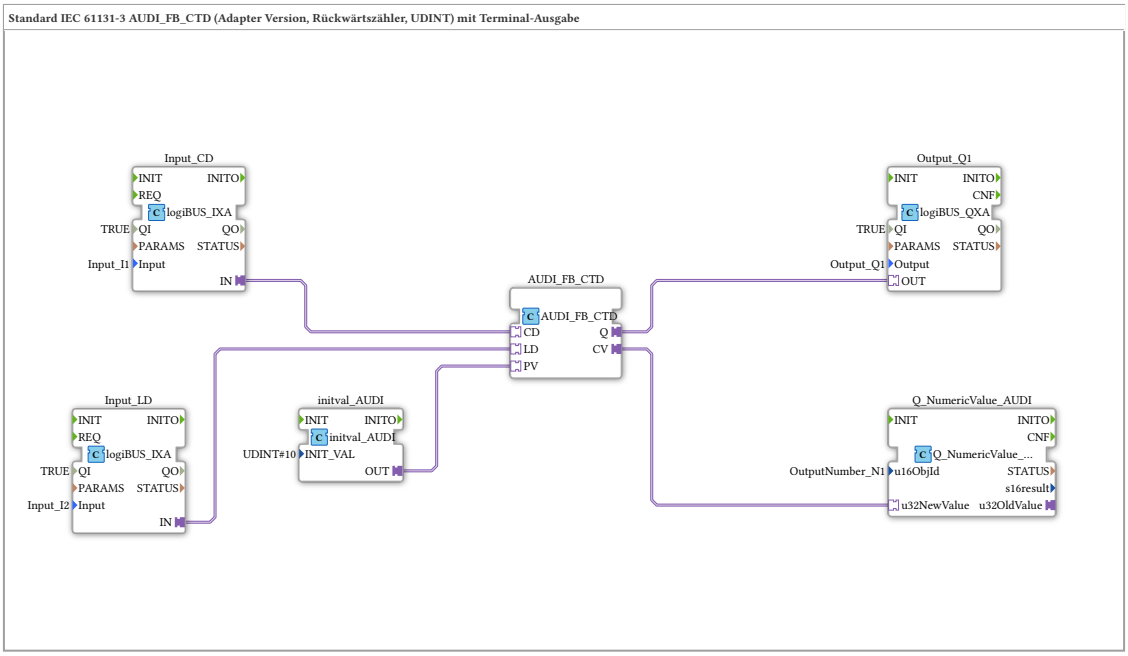
- Verständnis des IEC 61131-3 Zählers CTD (Abwärtszähler) mit LINT-Datentyp
- Umgang mit Adapter-Schnittstellen (ALI, LI, ALR)
- Konvertierung zwischen Datentypen (LINT → LREAL)
- Einbindung einer numerischen Terminalausgabe
- Erkennung von Ereignis- und Datenflüssen in einem ST-Netzwerk

Schwierigkeitsgrad: Fortgeschritten **Vorkenntnisse:** Grundlagen der 4diac-IDE, Umgang mit logiBUS-Bausteinen, Ereignisverkettung.

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

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

2.1.2.504 Uebung_218_AUDI: Standard IEC 61131-3 AUDI_FB_CTD (Adapter Version, Rückwärtszähler, UDINT) mit Terminal-Ausgabe



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung realisiert einen IEC 61131-3 konformen Rückwärtszähler (`AUDI_FB_CTD`) mit UDINT-Datentyp in der Adapter-Version. Der Zählerwert wird über einen Terminal-Baustein ausgegeben. Der Vorgabewert (Preset Value `PV`) wird durch einen separaten Baustein (`initval_AUDI`) auf 10 gesetzt. Die Übung demonstriert die Verwendung von Adapter-basierten Funktionsbausteinen und die Zusammenarbeit mit Ein-/Ausgangsmodulen sowie einer numerischen Terminalanzeige.

Verwendete Funktionsbausteine (FBs)

- **Input_CD**: Digitaler Eingang für „Count Down“
  - **Typ**: `logiBUS::io::DI::logiBUS_IXA`
  - **Parameter**: `QI = TRUE`, `Input = Input_I1`
  - **Erklärung**: Liefert den Zustand von `I1` als AX-Adapter `IN` an `AUDI_FB_CTD.CD`.
- **Input_LD**: Digitaler Eingang für „Load“

  - **Typ**: `logiBUS::io::DI::logiBUS_IXA`
  - **Parameter**: `QI = TRUE`, `Input = Input_I2`
  - **Erklärung**: Liefert den Zustand von `I2` als AX-Adapter `IN` an `AUDI_FB_CTD.LD`.
- **initval_AUDI**: Vorgabewert
  - **Typ**: `adapter::types::unidirectional::AUDI::initval::initval_AUDI`
  - **Parameter**: `INIT_VAL = 10`
  - **Erklärung**: Liefert beim Initialisieren den UDINT-Wert 10 als Adapter `OUT` an `AUDI_FB_CTD.PV`.
- **AUDI_FB_CTD**: Rückwärtszähler
  - **Typ**: `adapter::iec61131::counters::AUDI_FB_CTD`
  - **Parameter**: keine
  - **Erklärung**: Adapter-Version des IEC 61131-3 Abwärtszählers (CTD, UDINT). Eingänge als Adapter: `CD` (Count Down, AX), `LD` (Load, AX) und `PV` (Preset Value, AUDI). Ausgänge als Adapter: `Q` (AX, TRUE bei Zählerstand 0) und `CV` (aktueller Zählerstand, AUDI). Die Ausgänge werden nur bei Änderung gemeldet.
- **Output_Q1**: Digitaler Ausgang
  - **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
  - **Parameter**: `QI = TRUE`, `Output = Output_Q1`
  - **Erklärung**: Gibt `AUDI_FB_CTD.Q` als binäres Signal an `Q1` aus.
- **Q_NumericValue_AUDI**: Terminal-Ausgabe
  - **Typ**: `isobus::UT::Q::Q_NumericValue_AUDI`
  - **Parameter**: `u16ObjId = OutputNumber_N1`
  - **Erklärung**: Empfängt den Zählerstand (`CV`) über einen `AUDI`-Adapter (`u32NewValue`) und zeigt ihn im Objekt `OutputNumber_N1` an.

Programmablauf und Verbindungen

Das Netzwerk enthält ausschließlich Adapterverbindungen (keine EventConnections):

1. `Input_CD.IN` ⟶ `AUDI_FB_CTD.CD`: `I1` zählt den Zähler herunter.
2. `Input_LD.IN` ⟶ `AUDI_FB_CTD.LD`: `I2` lädt den Zähler mit dem Preset-Wert.
3. `initval_AUDI.OUT` ⟶ `AUDI_FB_CTD.PV`: der Preset-Wert ist 10.
4. `AUDI_FB_CTD.Q` ⟶ `Output_Q1.OUT`: `Q1` leuchtet, sobald der Zählerstand 0 erreicht hat.
5. `AUDI_FB_CTD.CV` ⟶ `Q_NumericValue_AUDI.u32NewValue`: der aktuelle Zählerstand wird am Terminal angezeigt.

Lernziele:

- Verständnis des IEC 61131-3 Abwärtszählers (CTD) als Adapterbaustein.
- Umgang mit Adapterverbindungen in 4diac.
- Vorgabe von Zählerwerten über einen `initval`-Baustein.
- Visualisierung von Werten auf einem Terminal.

Schwierigkeitsgrad: Mittel

Benötigte Vorkenntnisse: Grundlagen der SPS-Programmierung, Umgang mit 4diac-IDE, Verständnis von Ereignis- und Datenflüssen.

Durchführung:

- Importieren Sie die Übung in die 4diac-IDE.
- Weisen Sie die Ein-/Ausgänge den entsprechenden realen oder simulierten Hardware-Pins zu.
- Starten Sie die Anwendung und beobachten Sie das Verhalten: Drücken von `I2` lädt den Zähler mit 10. Jedes Drücken von `I1` zählt um 1 herunter, bis 0 erreicht ist - dann leuchtet `Q1`. Der aktuelle Zählerstand wird im Terminal angezeigt.

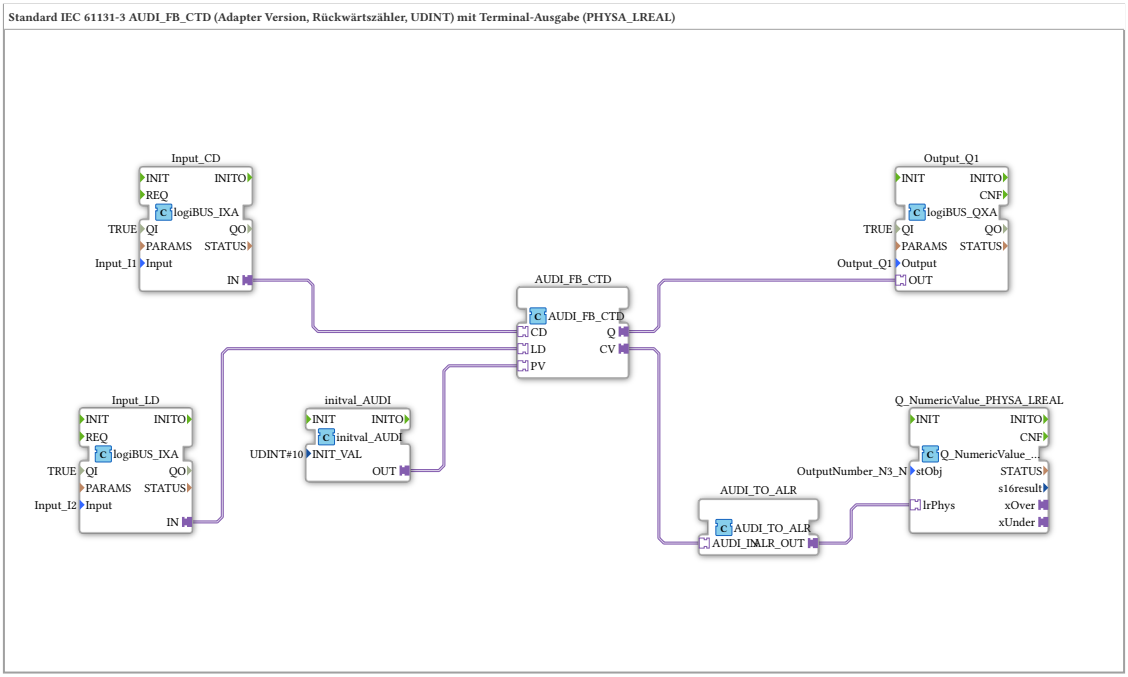
Zusammenfassung

Die Übung 218 demonstriert einen vollständigen IEC 61131-3 Abwärtszähler mit UDINT-Datentyp, der über Adapterverbindungen mit digitalen Ein- und Ausgängen sowie einer Terminalausgabe kommuniziert. Der Preset-Wert wird durch einen separaten `initval`-Baustein bereitgestellt.

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

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

2.1.2.505 Uebung_218b_ALR: Standard IEC 61131-3 AUDI_FB_CTD (Adapter Version, Rückwärtszähler, UDINT) mit Terminal-Ausgabe (PHYSA_LREAL)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung realisiert einen **Rückwärtszähler (CTD)** nach IEC 61131-3 mit einem `AUDI_FB_CTD` (Adapter-Version, Datentyp UDINT). Der Zähler wird über zwei digitale Eingänge gesteuert (`I1` = Count Down, `I2` = Load); der Preset-Wert 10 kommt von einem `initval_AUDI`. Der aktuelle Zählerstand wird über `AUDI_TO_ALR` in einen LREAL-Wert gewandelt und mit `Q_NumericValue_PHYSA_LREAL` auf dem Terminal ausgegeben. Zusätzlich zeigt ein digitaler Ausgang (`Output_Q1`) den Zählerausgang `Q` an.

Zwei Kommentare im Netzwerk weisen darauf hin: „hier sind negative Werte möglich!“ (bei der LREAL-Ausgabe) und „hier gegebenenfalls einen AX_D_FF einbauen, damit die Events reduziert werden.“ (am Ausgang `Q`).

Verwendete Funktionsbausteine (FBs)

- **Input_CD**: Eingang für den Abwärtszählimpuls
  - **Typ**: `logiBUS::io::DI::logiBUS_IXA`
  - **Parameter**: `QI = TRUE`, `Input = Input_I1`
  - **Erklärung**: Stellt den Zustand von `I1` als AX-Adaptersignal bereit; es speist `AUDI_FB_CTD.CD`.
- **Input_LD**: Eingang für das Laden
  - **Typ**: `logiBUS::io::DI::logiBUS_IXA`
  - **Parameter**: `QI = TRUE`, `Input = Input_I2`
  - **Erklärung**: Stellt den Zustand von `I2` als AX-Adaptersignal bereit; es speist `AUDI_FB_CTD.LD`.
- **initval_AUDI**: Preset-Wert
  - **Typ**: `adapter::types::unidirectional::AUDI::initval::initval_AUDI`
  - **Parameter**: `INIT_VAL = 10`
  - **Erklärung**: Stellt den konstanten Wert 10 als AUDI-Signal bereit.
- **AUDI_FB_CTD**: Rückwärtszähler
  - **Typ**: `adapter::iec61131::counters::AUDI_FB_CTD`
  - **Parameter**: keine
  - **Erklärung**: Zählt über den Adapter `CD` abwärts, lädt über `LD` den Preset-Wert `PV` (UDINT) und gibt den aktuellen Zählerstand am Adapter `CV` sowie den Zählerausgang am Adapter `Q` (laut Bausteinreferenz: TRUE, wenn der Zählerstand 0 erreicht ist) aus.
- **Output_Q1**: logiBUS Digitalausgang
  - **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
  - **Parameter**: `QI = TRUE`, `Output = Output_Q1`

  - **Erklärung**: Gibt den Zählerausgang `Q` am physischen Ausgang `Q1` aus.
- **AUDI_TO_ALR**: Konvertierungsadapter
  - **Typ**: `adapter::conversion::unidirectional::AUDI_TO_ALR`
  - **Parameter**: keine
  - **Erklärung**: Wandelt den Zählerstand (UDINT, Adapter `AUDI_IN`) in einen LREAL-Wert (Adapter `ALR_OUT`) um.
- **Q_NumericValue_PHYSA_LREAL**: VT-Anzeige
  - **Typ**: `isobus::UT::Q::Q_NumericValue_PHYSA_LREAL`
  - **Parameter**: `stObj = OutputNumber_N3_N`
  - **Erklärung**: Zeigt den physikalischen LREAL-Wert vom Adapter `lrPhys` auf dem im Objektpool-Eintrag `stObj` angegebenen VT-Objekt an.

Programmablauf und Verbindungen

1. `Input_CD.IN` ⟶ `AUDI_FB_CTD.CD` (AdapterConnection): Der Zustand von `I1` ist der Zähleingang.
2. `Input_LD.IN` ⟶ `AUDI_FB_CTD.LD`: Der Zustand von `I2` löst das Laden aus.
3. `initval_AUDI.OUT` ⟶ `AUDI_FB_CTD.PV`: Der Preset-Wert 10 wird dem Zähler vorgegeben.
4. `AUDI_FB_CTD.Q` ⟶ `Output_Q1.OUT`: Der Zählerausgang wird auf `Q1` gelegt.
5. `AUDI_FB_CTD.CV` ⟶ `AUDI_TO_ALR.AUDI_IN`: Der Zählerstand wird in LREAL gewandelt.
6. `AUDI_TO_ALR.ALR_OUT` ⟶ `Q_NumericValue_PHYSA_LREAL.lrPhys`: Der Wert wird auf dem Terminal ausgegeben.

Das Netzwerk enthält keine Ereignisverbindungen; alle Verbindungen sind Adapterverbindungen.
Lernziele:

- Aufbau und Parametrierung eines Adapter-basierten CTD-Funktionsbausteins
- Einbindung digitaler Ein-/Ausgänge über logiBUS-Adapter
- Datenkonvertierung (UDINT ⟶ LREAL) für Ausgabezwecke
- Erkennung von Problemen bei hohen Ereignisraten und Lösungsansätze (`AX_D_FF`)

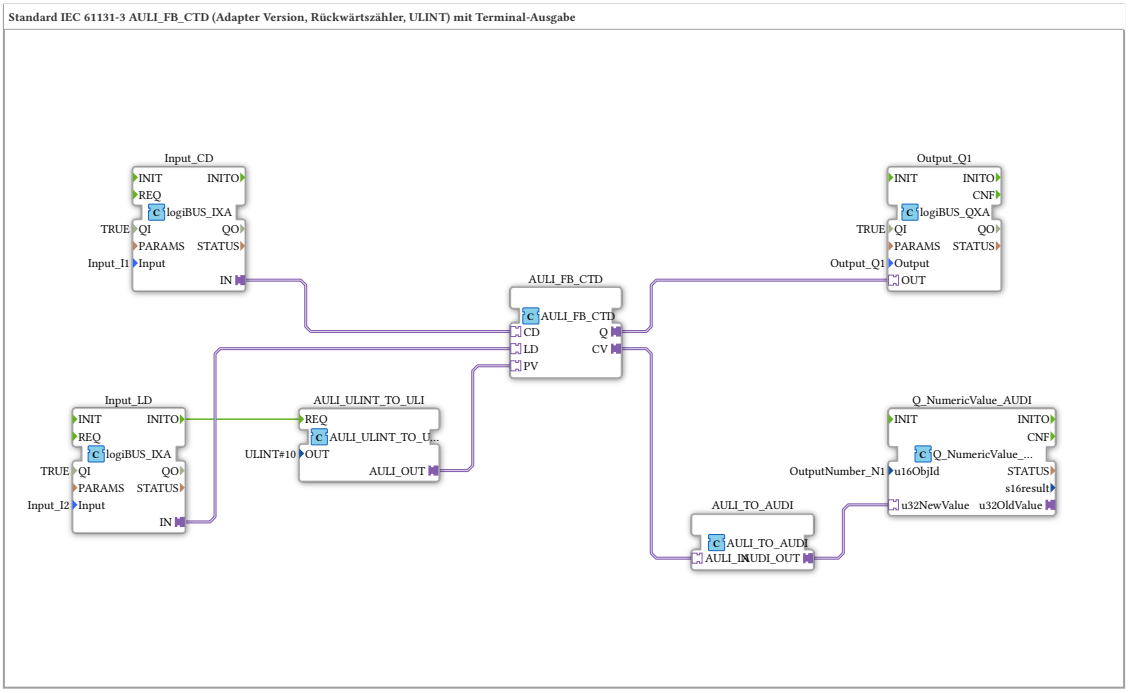
Zusammenfassung

Die Übung `Uebung_218b_ALR` vermittelt den Umgang mit einem IEC-61131-3-Rückwärtszähler in der 4diac-IDE. Der Zähler wird über zwei digitale Eingänge gesteuert (Zählen und Laden), sein Preset-Wert stammt von einem `initval_AUDI`, und er gibt seinen Status sowie den aktuellen Wert aus. Die Besonderheit liegt in der Umwandlung des UDINT-Zählwertes in eine LREAL-Terminalausgabe.

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

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

2.1.2.506 Uebung_219_AULI: Standard IEC 61131-3 AULI_FB_CTD (Adapter Version, Rückwärtszähler, ULINT) mit Terminal-Ausgabe



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung implementiert einen **Rückwärtszähler (Count Down)** nach IEC 61131-3 (Typ `CTD`) im **Adapter-Format**. Der Zähler arbeitet mit dem Datentyp **ULINT** (Unsigned Long Integer) und gibt den aktuellen Zählerstand über ein Terminal (z. B. System Display) aus. Zusätzlich wird ein digitaler Ausgang gesetzt, sobald der Zählerstand Null erreicht.

Die gesamte Funktionalität ist in einer SubApplikation (SubApp) gekapselt, die über logiBUS-Ein- und Ausgänge mit der Hardware verbunden ist.

Verwendete Funktionsbausteine (FBs)

Die SubApp enthält folgende interne Funktionsbausteine:

FB: AULI_FB_CTD

- **Typ**: `adapter::iec61131::counters::AULI_FB_CTD`
- **Beschreibung**: Kernbaustein – der eigentliche Rückwärtszähler. Er besitzt die für einen CTD typischen Schnittstellen:
  - **Ereigniseingänge**: – (wird automatisch durch die Adapterverbindungen gesteuert)
  - **Dateneingänge**: `CD` (Count-Down-Impuls), `LD` (Laden des Startwerts), `PV` (Preset Value, ULINT)
  - **Datenausgänge**: `Q` (Ausgangssignal = TRUE wenn Zählerstand = 0), `CV` (aktueller Zählerstand, ULINT)
- **Parameter**: keine zusätzlichen Parameter

FB: AULI_ULINT_TO_ULI

- **Typ**: `adapter::conversion::unidirectional::AULI_ULINT_TO_ULI`
- **Beschreibung**: Konvertiert einen konstanten ULINT-Wert (10) in das Format `ULI` (benötigt vom CTD-Baustein).
- **Parameter**: `OUT` = `ULINT#10` (Festwert, der als Preset-Wert dient)
- **Ereigniseingang**: `REQ` (wird durch `Input_LD.INITO` getriggert)
- **Datenausgang**: `AULI_OUT` (verbunden mit `AULI_FB_CTD.PV`)

FB: Input_CD

- **Typ**: `logiBUS::io::DI::logiBUS_IXA`

- **Beschreibung**: Digitaler Eingang für das Signal `CD` (Count Down). Liest den physischen Eingang `Input_I1`.
- **Parameter**: `QI` = `TRUE` (Initialisierung), `Input` = `Input_I1`
- **Adapterausgang**: `IN` (verbunden mit `AULI_FB_CTD.CD`)

FB: Input_LD

- **Typ**: `logiBUS::io::DI::logiBUS_IXA`
- **Beschreibung**: Digitaler Eingang für das Signal `LD` (Load). Liest den physischen Eingang `Input_I2`.
- **Parameter**: `QI` = `TRUE`, `Input` = `Input_I2`
- **Adapterausgang**: `IN` (verbunden mit `AULI_FB_CTD.LD`)
- **Ereignisausgang**: `INITO` (triggert die Umwandlung des Preset-Werts)

FB: Output_Q1

- **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
- **Beschreibung**: Digitaler Ausgang. Wird aktiv, sobald der Zählerstand Null erreicht (Q-Signal des CTD).
- **Parameter**: `QI` = `TRUE`, `Output` = `Output_Q1`
- **Adaptereingang**: `OUT` (verbunden mit `AULI_FB_CTD.Q`)

FB: AULI_TO_AUDI

- **Typ**: `adapter::conversion::unidirectional::AULI_TO_AUDI`
- **Beschreibung**: Konvertiert den aktuellen Zählerstand (`CV`, Typ AULI) in den Typ AUDI, der für die Terminalausgabe benötigt wird.
- **Adaptereingang**: `AULI_IN` (verbunden mit `AULI_FB_CTD.CV`)
- **Adapterausgang**: `AUDI_OUT` (verbunden mit `Q_NumericValue_AUDI.u32NewValue`)

FB: Q_NumericValue_AUDI

- **Typ**: `isobus::UT::Q::Q_NumericValue_AUDI`
- **Beschreibung**: Baustein zur Anzeige eines numerischen Werts im Terminal. Erhält den aktuellen Zählerstand und zeigt ihn auf dem zugeordneten Ausgabeobjekt an.
- **Parameter**: `u16ObjId` = `OutputNumber_N1` (Terminal-Ausgabeobjekt)
- **Dateneingang**: `u32NewValue` (verbunden mit `AULI_TO_AUDI.AUDI_OUT`)

Programmablauf und Verbindungen

1. **Initialisierung**: Beim Start wird durch das Ereignis `Input_LD.INITO` der Baustein `AULI_ULINT_TO_ULI` getriggert. Dieser liefert den festen Preset-Wert 10 (ULINT) an den Dateneingang `PV` des Zählers.
2. **Zähler laden**: Ein positiver Flanke am digitalen Eingang `Input_I2` (LD) lädt den Zähler mit dem Preset-Wert. Der aktuelle Zählerstand wird auf 10 gesetzt.
3. **Rückwärts zählen**: Jeder positive Flanke am digitalen Eingang `Input_I1` (CD) verringert den Zählerstand um 1, solange der Stand größer als 0 ist.
4. **Ausgabe des Zählerstandes**: Der aktuelle Zählerstand (`CV`) wird über die Konvertierungskette `AULI_TO_AUDI` → `Q_NumericValue_AUDI` auf das Terminal ausgegeben (Objekt `OutputNumber_N1`).
5. **Signal bei Null**: Sobald der Zählerstand den Wert 0 erreicht, setzt der CTD-Baustein den Ausgang `Q` auf TRUE. Dies schaltet den digitalen Ausgang `Output_Q1` ein.

Zusammenfassung der Verbindungen (Adapter- und Ereignis-Connections):

Quelle	Ziel	Art
`Input_CD.IN`	`AULI_FB_CTD.CD`	Adapter
`Input_LD.IN`	`AULI_FB_CTD.LD`	Adapter
`AULI_FB_CTD.Q`	`Output_Q1.OUT`	Adapter
`AULI_FB_CTD.CV`	`AULI_TO_AUDI.AULI_IN`	Adapter
`AULI_TO_AUDI.AUDI_OUT`	`Q_NumericValue_AUDI.u32NewValue`	Adapter
`AULI_ULINT_TO_ULI.AULI_OUT`	`AULI_FB_CTD.PV`	Adapter
`Input_LD.INITO`	`AULI_ULINT_TO_ULI.REQ`	Ereignis

Zusammenfassung

Lernziele dieser Übung:

- Aufbau und Funktionsweise eines IEC 61131-3 Rückwärtszählers (CTD) im Adapter-Format verstehen.
- Einbindung digitaler Ein- und Ausgänge über logiBUS.
- Konvertierung von Datentypen für die Kommunikation zwischen Bausteinen.
- Ausgabe von Prozesswerten auf einem Terminal (isobus-basiert).

Schwierigkeitsgrad: Mittel

Vorkenntnisse: Grundlagen der 4diac-IDE, Umgang mit logiBUS-I/O-Bausteinen, einfache Datenkonvertierung.

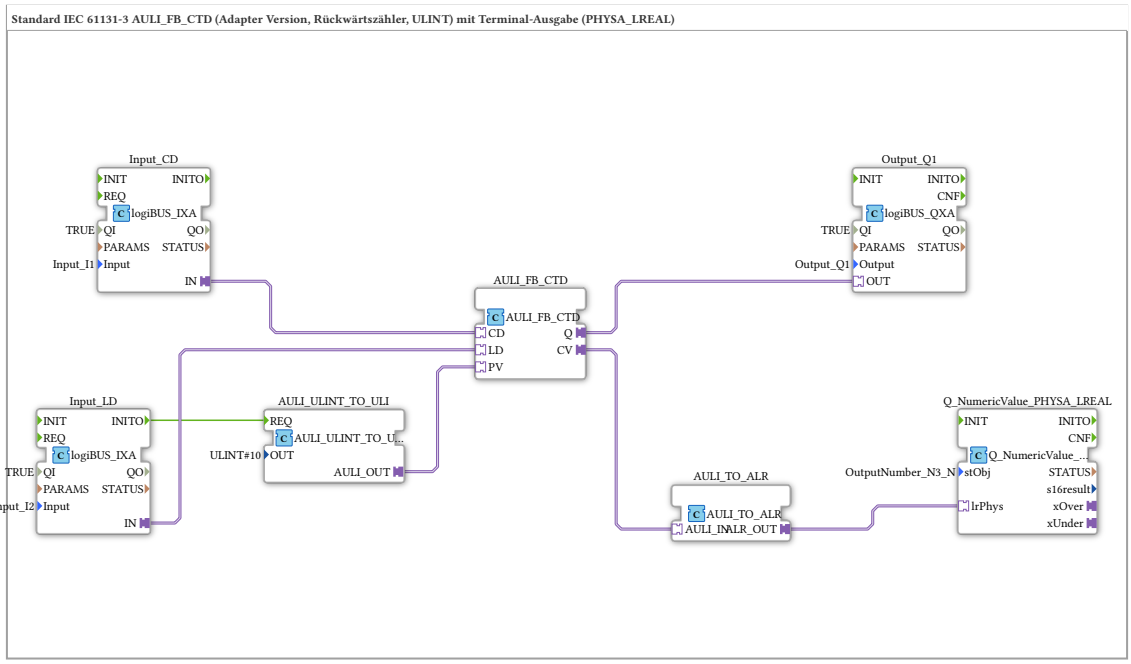
Starten der Übung:

- Die SubApp kann in ein 4diac-Projekt eingefügt und auf eine entsprechende Steuerung (mit logiBUS-Anbindung und Terminal) geladen werden.
- Die Eingänge `Input_I1` (CD) und `Input_I2` (LD) müssen mit Tastern oder einem Signalgenerator verbunden sein.
- Der Ausgang `Output_Q1` und das Terminalobjekt `OutputNumber_N1` zeigen die Ergebnisse an.

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

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

2.1.2.507 Uebung_219b_ALR: Standard IEC 61131-3 AULI_FB_CTD (Adapter Version, Rückwärtszähler, ULINT) mit Terminal-Ausgabe (PHYSA_LREAL)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung implementiert einen Rückwärtszähler (CTD – Count Down) gemäß IEC 61131-3 als Adapter-Version. Der Zähler verarbeitet ULINT-Werte und gibt das aktuelle Zählergebnis über eine Terminalausgabe (PHYSA_LREAL) aus. Zusätzlich wird ein digitaler Ausgang gesetzt, wenn der Zählwert Null erreicht.

Die Übung demonstriert den Einsatz von:

- Adapter-basierten Funktionsbausteinen für Zähler und Konvertierungen
- physikalischen Ein- und Ausgängen (logiBUS)
- Formatierung und Ausgabe numerischer Werte auf einem Terminal

Verwendete Funktionsbausteine (FBs)

AULI_FB_CTD

- **Typ**: `adapter::iec61131::counters::AULI_FB_CTD`
- **Beschreibung**: Rückwärtszähler (Count Down) für ULINT-Datentyp in Adapter-Ausführung. Er besitzt die Ereigniseingänge `CD` (Count Down), `LD` (Load) sowie die Datenausgänge `Q` (Null erreicht) und `CV` (aktueller Zählwert).
- **Parameter**: keine
- **Ereignisse**: nicht direkt verbunden (Ereignisse der Eingänge werden über Adapterverbindungen gesteuert)
- **Daten**:
  - `PV` (Preset Value) erhält den Startwert über `AULI_ULINT_TO_ULI`
  - `CV` (Current Value) wird auf `AULI_TO_ALR` gegeben

AULI_ULINT_TO_ULI

- **Typ**: `adapter::conversion::unidirectional::AULI_ULINT_TO_ULI`
- **Beschreibung**: Konvertiert ULINT in ULINT (hier eher als Initialisierung genutzt). Der Parameter `OUT` wird auf `ULINT#10` gesetzt, d.h. der Zähler startet bei 10.
- **Parameter**:
  - `OUT = ULINT#10`
- **Ereigniseingang**: `REQ` (von `Input_LD.INIT0`)
- **Datenausgang**: `AULI_OUT` ⟶ `PV` des Zählers

Input_CD (logiBUS_IXA)

- **Typ**: `logiBUS::io::DI::logiBUS_IXA`
- **Beschreibung**: Digitaler Eingang für das Count-Down-Signal (Tasteneingang `Input_I1`). Aktiv bei TRUE.
- **Parameter**:
  - `QI = TRUE`
  - `Input = Input_I1`

Input_LD (logiBUS_IXA)

- **Typ**: `logiBUS::io::DI::logiBUS_IXA`
- **Beschreibung**: Digitaler Eingang für das Load-Signal (Tasteneingang `Input_I2`). Aktiv bei TRUE.
- **Parameter**:
  - `QI = TRUE`
  - `Input = Input_I2`

Output_Q1 (logiBUS_QXA)

- **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
- **Beschreibung**: Digitaler Ausgang, der gesetzt wird, wenn der Zählwert Null erreicht (`Q` des Zählers).
- **Parameter**:
  - `QI = TRUE`
  - `Output = Output_Q1`

AULI_TO_ALR

- **Typ**: `adapter::conversion::unidirectional::AULI_TO_ALR`
- **Beschreibung**: Konvertiert den aktuellen ULINT-Zählwert in einen LREAL-Wert für die physische Ausgabe (AR).
- **Ereignisse**: keine direkte Ereignisverbindung
- **Daten**:
  - `AULI_IN` ⟵ `CV` des Zählers
  - `ALR_OUT` ⟶ `Q_NumericValue_PHYSA_LREAL.lrPhys`

Q_NumericValue_PHYSA_LREAL

- **Typ**: `isobus::UT::Q::Q_NumericValue_PHYSA_LREAL`
- **Beschreibung**: Gibt den übergebenen LREAL-Wert als formatierten numerischen String auf einem Terminal aus (Objekt `OutputNumber_N3`).
- **Parameter**:

  - `stObj = OutputNumber_N3`
- **Daten**:
  - `lrPhys` ⟵ `ALR_OUT` von `AULI_TO_ALR`

Programmablauf und Verbindungen

Die folgende Beschreibung erläutert den Daten- und Ereignisfluss:

1. **Initialisierung** Beim Systemstart (oder nach einem Reset) wird das Ereignis `INITO` des Eingangsbausteins `Input_LD` ausgelöst. Dieses Ereignis triggert die Konvertierung `AULI_ULINT_TO_ULI` über deren `REQ`-Eingang. Der Baustein gibt daraufhin den Wert `ULINT#10` als Startwert an den `PV`-Eingang des Zählers `AULI_FB_CTD`.
2. **Load-Vorgang** Wird der Eingang `Input_LD` (Taster I2) betätigt, so lädt der Zähler den Preset-Wert (10) in seinen aktuellen Zählwert `CV`. Dies geschieht über die Adapterverbindung `Input_LD.IN` ⟶ `AULI_FB_CTD.LD`.
3. **Count-Down** Jede steigende Flanke am Eingang `Input_CD` (Taster I1) verringert den Zählwert um 1. Die Verbindung `Input_CD.IN` ⟶ `AULI_FB_CTD.CD` realisiert dies.
4. **Null-Erkennung** Sobald der Zählwert `CV` den Wert Null erreicht, setzt der Zähler seinen Ausgang `Q` auf TRUE. Dieser wird über die Verbindung `AULI_FB_CTD.Q` ⟶ `Output_Q1.OUT` an den physikalischen Ausgang `Output_Q1` weitergeleitet. Ein angeschlossenes Gerät (z.B. Lampe) signalisiert den Null-Zustand.
5. **Terminalausgabe** Der aktuelle Zählwert `CV` wird über die Konvertierungskette `AULI_FB_CTD.CV` ⟶ `AULI_TO_ALR.AULI_IN` ⟶ `AULI_TO_ALR.ALR_OUT` ⟶ `Q_NumericValue_PHYSA_LREAL.lrPhys` auf dem Terminal (Objekt `OutputNumber_N3`) angezeigt. Hierbei wird der ULINT-Wert in LREAL umgewandelt, sodass auch negative Werte (z.B. durch Überlauf) dargestellt werden können (siehe Hinweis-Kommentar in der Grafik).

Hinweise aus den Kommentaren:

- Im Bereich der Terminalausgabe sind negative Werte möglich (z.B. wenn der Zähler unter Null gezählt wird).
- Bei hohen Ereignisraten kann ein AX_D_FF (Verzögerungsglied) zwischengeschaltet werden, um die Anzahl der Terminalaktualisierungen zu reduzieren.

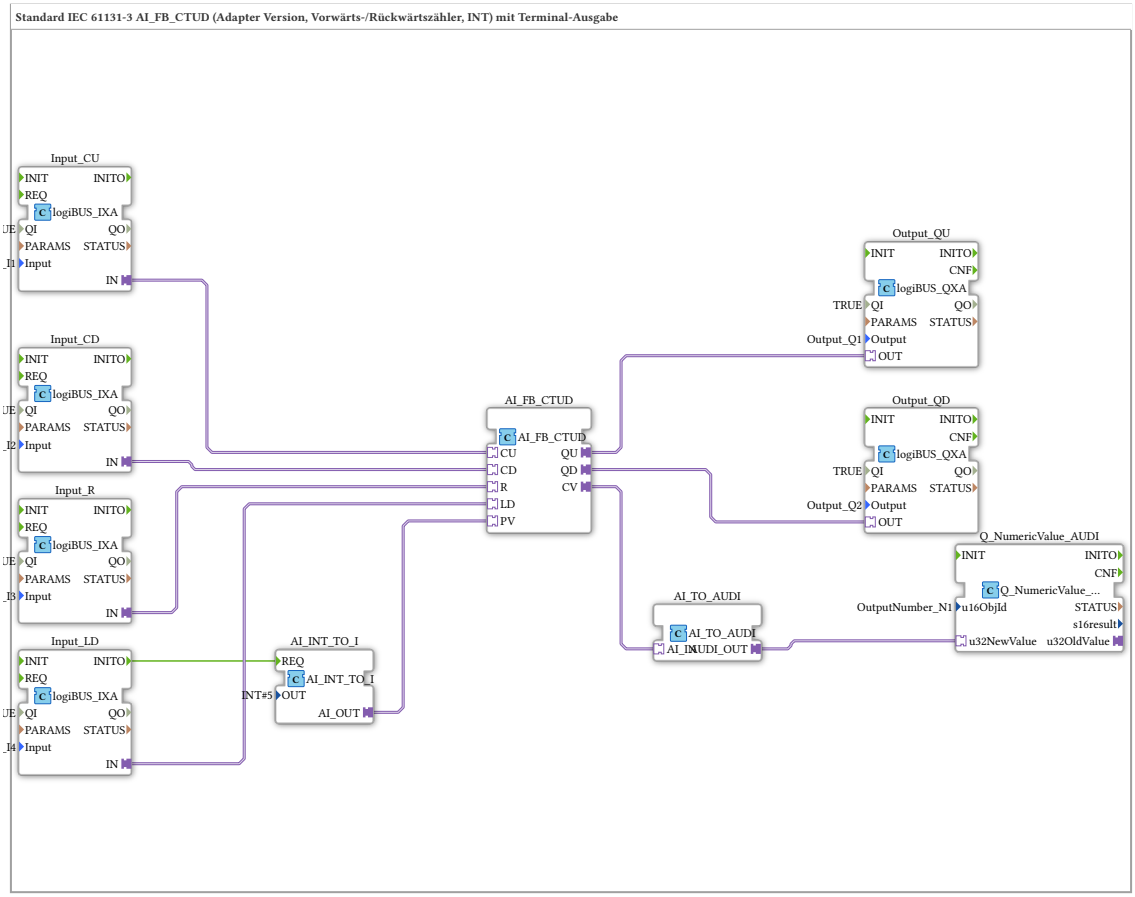
Zusammenfassung

Die Übung `Uebung_219b_ALR` realisiert einen IEC 61131-3 konformen Rückwärtszähler (CTD) in Adapter-Bauweise. Über digitale Eingänge wird der Zähler geladen und dekrementiert. Der aktuelle Zählwert wird auf einem Terminal ausgegeben, und ein digitaler Ausgang signalisiert das Erreichen des Nullwerts. Die Implementierung zeigt die Kombination von Standard-Zählerbausteinen, Konvertierungsadaptern und hardwarenaher Ein-/Ausgabe in der 4diac-IDE.

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

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

2.1.2.508 Uebung_220_AI: Standard IEC 61131-3 AI_FB_CTUD (Adapter Version, Vorwärts-/Rückwärtszähler, INT) mit Terminal-Ausgabe



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung implementiert einen **Vorwärts-/Rückwärtszähler nach IEC 61131-3 (CTUD)** als Adapter-Version für den Datentyp `INT`. Der aktuelle Zählerstand wird über ein Terminal (Numerikausgabe) ausgegeben. Die Steuerung erfolgt über vier digitale Eingänge (CU, CD, R, LD) und zwei digitale Ausgänge (QU, QD). Ein Konstantwert (INT#5) wird als Voreinstellwert (PV) geladen.

Verwendete Funktionsbausteine (FBs)

- **AI_FB_CTUD** (`adapter::iec61131::counters::AI_FB_CTUD`) Zentrale Zählerlogik: Ereignisgesteuerter Vorwärts-/Rückwärtszähler (CTUD) mit den Anschlüssen CU, CD, R, LD und PV. Liefert die Ausgänge QU (Überlauf), QD (Unterlauf) und CV (aktueller Zählerwert). Keine Parameter.
- **AI_INT_TO_I** (`adapter::conversion::unidirectional::AI_INT_TO_I`) Konvertiert einen konstanten `INT`-Wert (hier: 5) in ein Adapter-Format für den Voreinstellwert (PV). Parameter: `OUT = INT#5`.
- **Input_CU** (`logiBUS::io::DI::logiBUS_IXA`) Digitaler Eingang für das Signal „Count Up“ (Vorwärtszählen). Parameter: `QI = TRUE`, `Input = Input_I1`.
- **Input_CD** (`logiBUS::io::DI::logiBUS_IXA`) Digitaler Eingang für das Signal „Count Down“ (Rückwärtszählen). Parameter: `QI = TRUE`, `Input = Input_I2`.
- **Input_R** (`logiBUS::io::DI::logiBUS_IXA`) Digitaler Eingang für das Rücksetzsignal (Reset). Parameter: `QI = TRUE`, `Input = Input_I3`.
- **Input_LD** (`logiBUS::io::DI::logiBUS_IXA`) Digitaler Eingang für das Laden des Voreinstellwerts (Load). Parameter: `QI = TRUE`, `Input = Input_I4`.
- **Output_QU** (`logiBUS::io::DQ::logiBUS_QXA`) Digitaler Ausgang, der den Überlauf (QU) des Zählers signalisiert. Parameter: `QI = TRUE`, `Output = Output_Q1`.

- **Output_QD** (`logiBUS::io::DQ::logiBUS_QXA`) Digitaler Ausgang, der den Unterlauf (QD) des Zählers signalisiert. Parameter: `QI = TRUE`, `Output = Output_Q2`.
- **AI_TO_AUDI** (`adapter::conversion::unidirectional::AI_TO_AUDI`) Konvertiert den aktuellen Zählerwert (CV) vom Adapter-Format in ein numerisches Audio-Format (AUDI), das von der Ausgabe-Komponente verarbeitet werden kann. *Hinweis: Laut Kommentar im Quelltext ist dieser Baustein für negative Zahlen nicht geeignet.*
- **Q_NumericValue_AUDI** (`isobus::UT::Q::Q_NumericValue_AUDI`) Terminal-Ausgabebaustein zur Anzeige des Zählerwerts auf einem numerischen Display. Parameter: `u16ObjId = OutputNumber_N1`.

Programmablauf und Verbindungen

Der Ablauf wird durch Ereignisse gesteuert. Die Verbindungen sind wie folgt realisiert:

1. **Initialisierung & Voreinstellwert laden** Beim Start wird das Ereignis `Input_LD.INIT0` an `AI_INT_TO_I.REQ` weitergeleitet. Dadurch wird der konstante Wert `INT#5` über den Adapter `AI_INT_TO_I` an den PV-Eingang des Zählers `AI_FB_CTUD.PV` übergeben.
2. **Zählereingänge**
   - `Input_CU.IN` ⟶ `AI_FB_CTUD.CU` (Vorwärtszählen bei Flanke)
   - `Input_CD.IN` ⟶ `AI_FB_CTUD.CD` (Rückwärtszählen bei Flanke)
   - `Input_R.IN` ⟶ `AI_FB_CTUD.R` (Reset auf 0)
   - `Input_LD.IN` ⟶ `AI_FB_CTUD.LD` (Laden des Wertes von PV)
3. **Zählerausgänge**
   - `AI_FB_CTUD.QU` ⟶ `Output_QU.OUT` (Überlauf)
   - `AI_FB_CTUD.QD` ⟶ `Output_QD.OUT` (Unterlauf)
4. **Ausgabe des Zählerstandes** Der aktuelle Zählerwert `CV` wird über `AI_TO_AUDI` konvertiert und an den Ausgabebaustein `Q_NumericValue_AUDI.u32NewValue` gesendet. Dieser zeigt den Wert auf einer konfigurierten Terminalnummer (`u16ObjId = OutputNumber_N1`) an.

Hinweise aus dem Quelltext:

- Der Baustein `AI_TO_AUDI` unterstützt keine negativen Zahlen – daher können nur Zählwerte ≥ 0 korrekt angezeigt werden.
- Es wurde angemerkt, dass zur Reduktion der Ereignisrate ggf. flankengetriggerte D-Flipflops (z.B. `AX_D_FF`) zwischengeschaltet werden sollten, was jedoch in dieser Version nicht umgesetzt ist.

Zusammenfassung

Die Übung demonstriert den Einsatz eines IEC-61131-3-Zählers (CTUD) in einer 4diac-Adapterumgebung. Sie zeigt die Verknüpfung digitaler Ein-/Ausgänge, die Konvertierung von Datenformaten (`INT` ↔ Adapter ↔ AUDI) sowie die Ausgabe eines Zahlenwerts auf einem Terminal. Der Lerneffekt liegt im Verständnis ereignisgesteuerter Zähler, der Datenfluss-Konvertierung und der Fehlerbehandlung bei negativen Werten. Die Übung ist für Fortgeschrittene geeignet und setzt Grundkenntnisse in der 4diac-IDE und in IEC 61131-3 voraus.

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

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

2.1.2.509 Uebung_220b_AR: Standard IEC 61131-3 AI_FB_CTUD (Adapter Version, Vorwärts-/Rückwärtszähler, INT) mit Terminal-Ausgabe (PHYS)

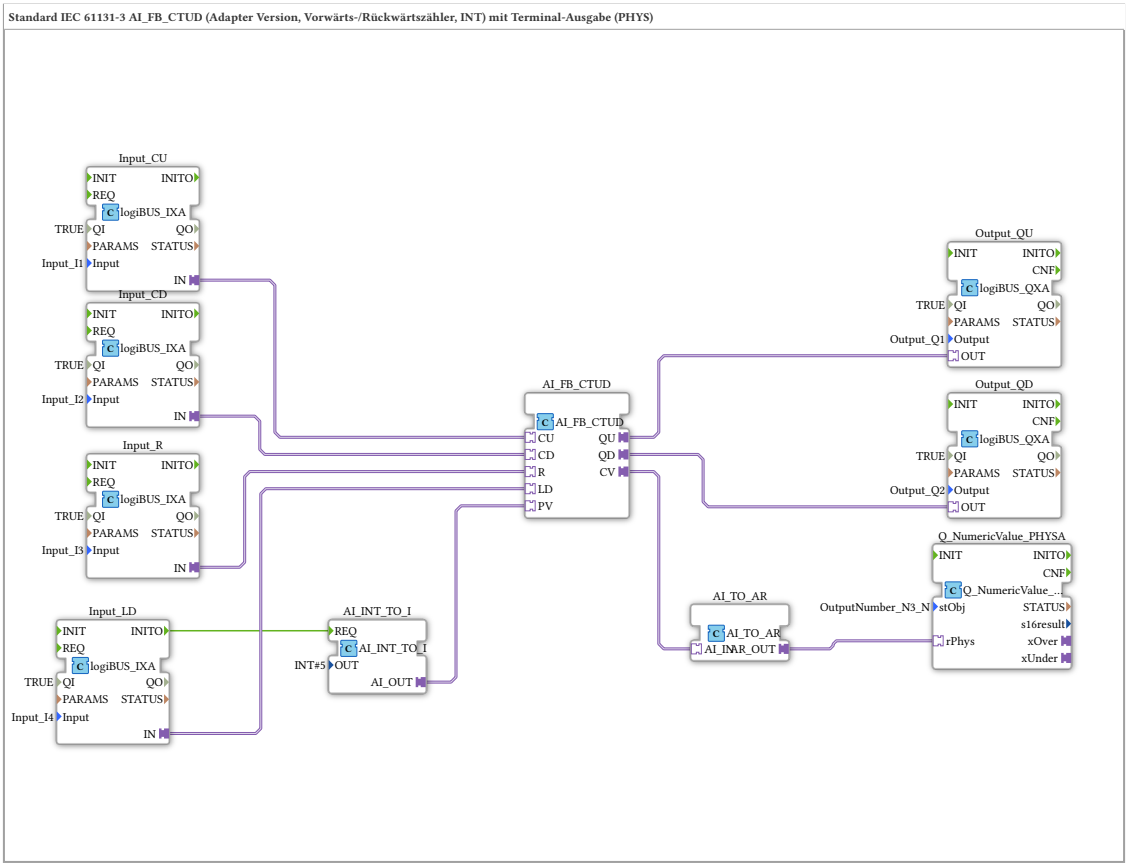


Bild nicht vorhanden

• – ○ • – ○ • – ○ •

Einleitung

Diese Übung realisiert einen Vorwärts-/Rückwärtszähler (CTUD) nach IEC 61131-3 in einer Adapter-Version. Der Zählerwert (Integer) wird über einen digitalen Preset-Wert (PV) initialisiert und kann über vier digitale Eingänge gesteuert werden. Der aktuelle Zählerstand wird über einen Analogausgang auf einem Terminal ausgegeben (physikalische Darstellung). Die Übung demonstriert die Verwendung von Adapter-basierten Funktionsbausteinen sowie die Umwandlung und Ausgabe von Zählerdaten.

Verwendete Funktionsbausteine (FBs)

- **AI_FB_CTUD**
  - **Typ**: `adapter::iec61131::counters::AI_FB_CTUD`
  - **Funktion**: Vorwärts-/Rückwärtszähler mit Adapter-Schnittstelle (Eingänge CU, CD, R, LD, PV; Ausgänge QU, QD, CV).
- **AI_INT_TO_I**
  - **Typ**: `adapter::conversion::unidirectional::AI_INT_TO_I`
  - **Parameter**: `OUT = INT#5` (konstanter Preset-Wert)
  - **Funktion**: Wandelt einen konstanten Integer-Wert in einen Adapter-Ausgang (AI_OUT) um, der als Preset (PV) an den Zähler geht.
- **Input_CU, Input_CD, Input_R, Input_LD**
  - **Typ**: jeweils `logiBUS::io::DI::logiBUS_IXA`
  - **Parameter**: `QI = TRUE` (Aktivierung des Eingangs); `Input` zugeordnet auf logiBUS-Digitaleingänge (Input_I1 bis Input_I4).
  - **Funktion**: Digital-Eingangsadapter für die Steuerung des Zählers (CU = Vorwärtszählimpuls, CD = Rückwärtszählimpuls, R = Reset, LD = Laden des Presetwerts).
- **Output_QU, Output_QD**

  - **Typ**: jeweils `logiBUS::io::DQ::logiBUS_QXA`
  - **Parameter**: `QI = TRUE`; `Output` zugeordnet auf logiBUS-Digitalausgänge (Output_Q1, Output_Q2).
  - **Funktion**: Digitalausgangsadapter für die Zählerausgänge QU (Vorwärtszähler erreicht) und QD (Rückwärtszähler erreicht).
- **AI_TO_AR**
  - **Typ**: `adapter::conversion::unidirectional::AI_TO_AR`
  - **Funktion**: Wandelt den Adapter-Ausgang (AI_IN) des Zählerwerts (CV) in einen analogen Adapter-Wert (AR_OUT) um.
- **Q_NumericValue_PHYSA**
  - **Typ**: `isobus::UT::Q::Q_NumericValue_PHYSA`
  - **Parameter**: `stObj = OutputNumber_N3` (vordefiniertes Terminal-Ausgabeobjekt)
  - **Funktion**: Gibt den analogen Zahlenwert physisch auf dem Terminal aus.

Programmablauf und Verbindungen

Die Ereignissteuerung erfolgt über einen einzigen Ereignispfad:

- Der Baustein **Input_LD** löst beim Initialisierungsereignis (`INIT0`) den Baustein **AI_INT_TO_I** aus (`REQ`), sodass dieser den konstanten Preset-Wert (INT#5) auf seinem Ausgang bereitstellt.

Die logischen Datenverbindungen (Adapterverbindungen) verknüpfen die Komponenten wie folgt:

- **Digitaleingänge zu Zähler:** `Input_CU.IN` ⟶ `AI_FB_CTUD.CU` (Vorwärtszählimpuls) `Input_CD.IN` ⟶ `AI_FB_CTUD.CD` (Rückwärtszählimpuls) `Input_R.IN` ⟶ `AI_FB_CTUD.R` (Reset) `Input_LD.IN` ⟶ `AI_FB_CTUD.LD` (Laden des Presetwerts)
- **Zählerausgänge zu Digitalausgängen:** `AI_FB_CTUD.QU` ⟶ `Output_QU.OUT` (Vorwärtszählererreicht) `AI_FB_CTUD.QD` ⟶ `Output_QD.OUT` (Rückwärtszählererreicht)
- **Zählerwert zu Terminalausgabe:** `AI_FB_CTUD.CV` ⟶ `AI_TO_AR.AI_IN` (Aktueller Zählerwert) `AI_TO_AR.AR_OUT` ⟶ `Q_NumericValue_PHYSA.rPhys` (Analogwert für Terminalanzeige)
- **Preset-Wert:** `AI_INT_TO_I.AI_OUT` ⟶ `AI_FB_CTUD.PV` (Setzt den Preset-Wert auf INT#5)

Ein Kommentar weist darauf hin, dass mit dem Zähler auch negative Werte möglich sind und dass bei Bedarf pro Ausgang ein AX_D_FF (D-Flipflop) eingebaut werden kann, um die Ereignisrate zu reduzieren. Der Baustein **AI_FB_CTUD** arbeitet als Zähler: Bei jedem steigenden Flanke an CU wird der Zähler erhöht, bei CD verringert, bei R auf 0 zurückgesetzt und bei LD auf den Wert von PV geladen. Die Ausgänge QU und QD werden aktiv, wenn der Zählerstand einen Schwellwert erreicht (typischerweise >0 für QU, <0 für QD).

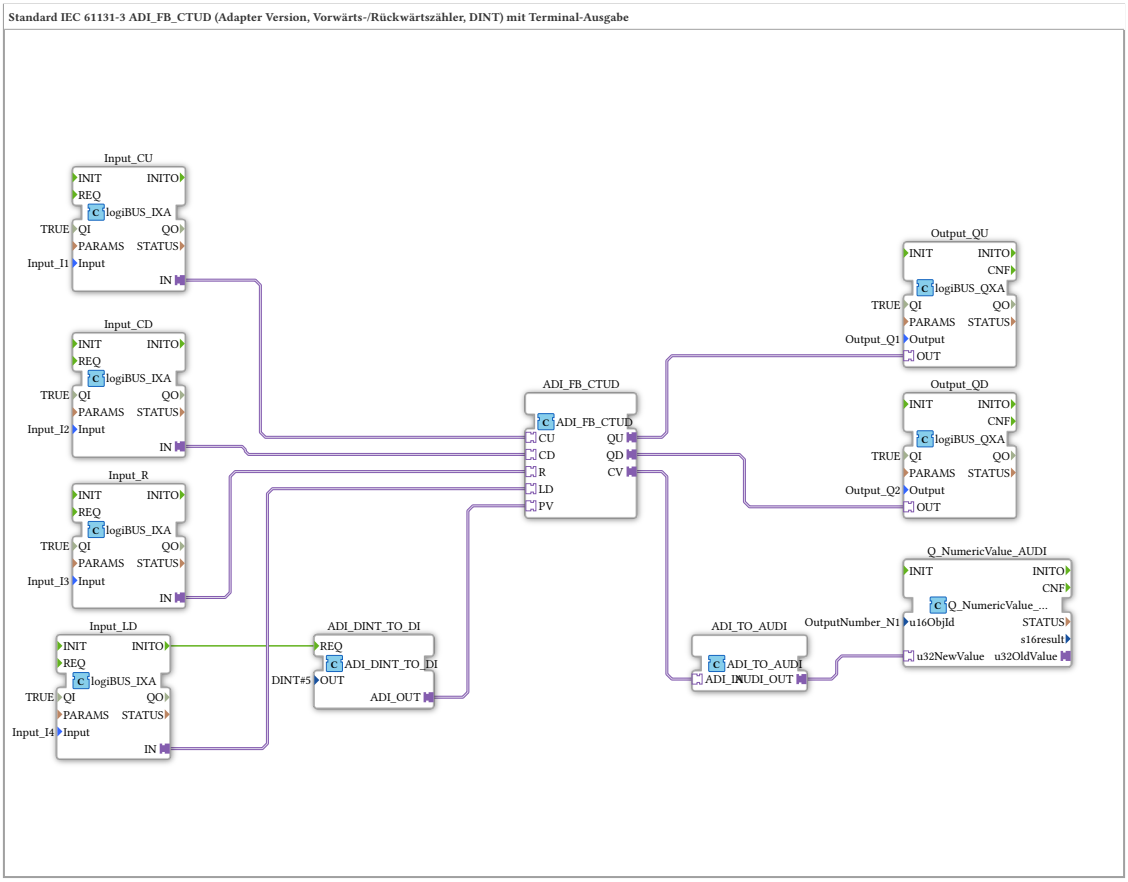
Zusammenfassung

Die Übung **Uebung_220b_AR** zeigt den Einsatz eines standardisierten IEC 61131-3 Zählers (CTUD) in einer Adapter-Architektur. Durch die Verbindung von digitalen Eingängen (Taster/Sensoren), Digitalausgängen und einer Terminalausgabe wird ein vollständiger Zählprozess mit Visualisierung abgebildet. Der Lernende versteht die Funktionsweise von Aufwärts-/Abwärtszählern, die Adapter-Kommunikation in 4diac sowie die Umwandlung von Datenformaten (INT über AI_TO_AR). Zusätzlich wird die Möglichkeit zur Ereignisreduzierung durch Flipflops thematisiert.

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

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

2.1.2.510 Uebung_221_ADI: Standard IEC 61131-3 ADI_FB_CTUD (Adapter Version, Vorwärts-/Rückwärtszähler, DINT) mit Terminal-Ausgabe



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung implementiert einen Vorwärts-/Rückwärtszähler (engl. *Counter Up/Down*, CTUD) nach IEC 61131-3 in einer Adapter-Version (ADI_FB_CTUD). Der Zähler arbeitet mit 32-Bit-Ganzzahlen (DINT) und gibt den aktuellen Zählwert über eine Terminalausgabe aus. Die Eingänge werden über logiBUS-Adapter eingelesen, die Ausgänge über logiBUS-Adapter gesteuert. Ein fester Preset-Wert von 5 wird bei der Initialisierung geladen.

Verwendete Funktionsbausteine (FBs)

- **ADI_FB_CTUD** (Typ: `adapter::iec61131::counters::ADI_FB_CTUD`)
  Hauptzählerbaustein. Er besitzt die Adapter-Schnittstellen CU (Vorwärtszählen), CD (Rückwärtszählen), R (Reset), LD (Load), PV (Preset Value) sowie die Ausgänge QU (Überlauf), QD (Unterlauf) und CV (aktueller Zählwert).
- **ADI_DINT_TO_DI** (Typ: `adapter::conversion::unidirectional::ADI_DINT_TO_DI`)
  Wandelt einen konstanten DINT-Wert (hier `DINT#5`) in eine DI-Adapter-Schnittstelle um. Parameter: `OUT = DINT#5`. Der Ausgang `ADI_OUT` wird mit dem PV-Eingang des Zählers verbunden.
- **logiBUS_IXA** (Eingangsadapter) – vier Instanzen:
  - `Input_CU`: Eingang für Vorwärtszählimpulse (angeschlossen an physischen Eingang `Input_I1`).
  - `Input_CD`: Eingang für Rückwärtszählimpulse (`Input_I2`).
  - `Input_R`: Reset-Eingang (`Input_I3`).
  - `Input_LD`: Load-Eingang (`Input_I4`).
    Alle haben den Parameter `QI = TRUE`.
- **logiBUS_QXA** (Ausgangsadapter) – zwei Instanzen:
  - `Output_QU`: Ausgang für Überlaufsignal (physischer Ausgang `Output_Q1`).

  - `Output_QD`: Ausgang für Unterlaufsignal (`Output_Q2`).
    Beide mit `QI = TRUE`.

- **ADI_TO_AUDI** (Typ: `adapter::conversion::unidirectional::ADI_TO_AUDI`)
  Wandelt den ADI-Datenstrom (vom CV des Zählers) in ein AUDI-Signal für die Terminalausgabe um. Hinweis: Diese Wandlung unterstützt keine negativen Zahlen (siehe Kommentar im Netzwerk).
- **Q_NumericValue_AUDI** (Typ: `isobus::UT::Q::Q_NumericValue_AUDI`)
  Terminalausgabe-Baustein. Parameter: `u16ObjId = OutputNumber_N1`. Zeigt den empfangenen numerischen Wert (Zählwert) auf dem Terminal an.

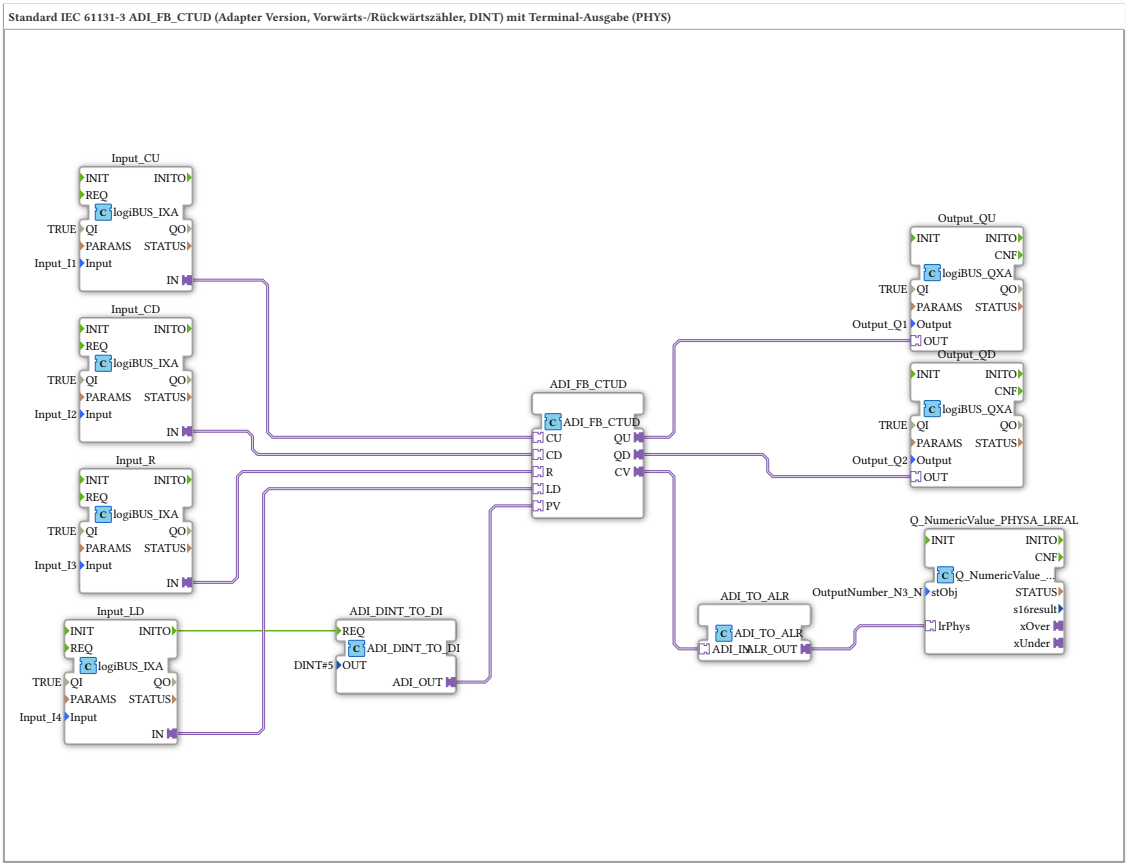
Programmablauf und Verbindungen

1. **Initialisierung**: Beim Start wird das INITO-Ereignis von `Input_LD` genutzt, um den Baustein `ADI_DINT_TO_DI` zu triggern. Dieser gibt daraufhin den festen Wert `DINT#5` als ADI-Schnittstelle an den PV-Eingang des Zählers `ADI_FB_CTUD` weiter. Somit wird der Preset-Wert auf 5 gesetzt.
2. **Zählbetrieb**:
   - Jeder positive Flanke an `Input_CU` (Vorwärtszählimpuls) erhöht den Zählerstand um 1.
   - Jede positive Flanke an `Input_CD` verringert den Zählerstand um 1.
   - Bei einem positiven Signal an `Input_R` wird der Zähler auf 0 zurückgesetzt.
   - Bei einem positiven Signal an `Input_LD` wird der Zähler auf den aktuellen Preset-Wert (hier 5) gesetzt.
3. **Ausgänge**:
   - Der Ausgang `Output_QU` (Überlauf) wird aktiv, wenn der Zähler den maximalen DINT-Wert erreicht und ein weiterer Vorwärtsimpuls erfolgt.
   - Der Ausgang `Output_QD` (Unterlauf) wird aktiv, wenn der Zähler den minimalen DINT-Wert erreicht und ein weiterer Rückwärtsimpuls erfolgt.
   - Der aktuelle Zählwert (CV) wird über `ADI_TO_AUDI` und `Q_NumericValue_AUDI` auf einem Terminal (Objekt-ID `OutputNumber_N1`) ausgegeben.
4. **Datenverbindungen** (Adapter Connections):
   - Die Adapter-Eingänge `CU`, `CD`, `R`, `LD` des Zählers werden direkt mit den entsprechenden logiBUS-Eingängen verbunden.
   - Die Adapter-Ausgänge `QU` und `QD` des Zählers werden mit den logiBUS-Ausgängen verbunden.
   - Der Adapter-Ausgang `CV` (Zählwert) wird an `ADI_TO_AUDI.ADI_IN` angeschlossen.
   - Der AUDI-Ausgang von `ADI_TO_AUDI` geht an `Q_NumericValue_AUDI.u32NewValue`.
   - Der feste Preset-Wert von `ADI_DINT_TO_DI` wird an `ADI_FB_CTUD.PV` übertragen.
5. **Hinweise**:
   - Die verwendete Konvertierung `ADI_TO_AUDI` kann keine negativen Zahlen darstellen (siehe Kommentar im Netzwerk). Sollte der Zähler negative Werte annehmen, ist die Terminalausgabe fehlerhaft.
   - Bei hochfrequenten Ereignissen kann es sinnvoll sein, je einen AX_D_FF-Baustein (Ereignis-Flipflop) zwischenzuschalten, um die Ereignisrate zu reduzieren (siehe Kommentar).

Zusammenfassung

Die Übung 221 demonstriert den Einsatz eines IEC-61131-3-konformen Aufwärts-/Abwärtszählers in der Adapter-Version. Der Zähler wird über physische Eingänge gesteuert und gibt seinen aktuellen Wert sowie Überlauf-/Unterlaufsignale auf Ausgänge und ein Terminal aus. Ein fester Preset-Wert wird initial geladen. Die Übung zeigt die Integration von Standard-FBs mit logiBUS-E/A und Terminalausgabe sowie die Datenkonvertierung zwischen verschiedenen Adapter-Schnittstellen. Hinweise auf Einschränkungen (keine negativen Zahlen) und Optimierungsmöglichkeiten (Ereignisreduktion) sind vorhanden.

2.1.2.511 Uebung_221b_ALR: Standard IEC 61131-3 ADI_FB_CTUD (Adapter Version, Vorwärts-/Rückwärtszähler, DINT) mit Terminal-Ausgabe (PHYS)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung implementiert einen Vorwärts-/Rückwärtszähler (Up/Down Counter) nach IEC 61131-3 (Typ `ADI_FB_CTUD`). Der Zähler wird über digitale Eingänge gesteuert und gibt den aktuellen Zählerstand sowohl über digitale Ausgänge (als Grenzwert-Signale) als auch über eine Terminal-Ausgabe (physikalischer Wert) aus. Der Zählbereich arbeitet mit 32-Bit Ganzzahlen (DINT), wobei auch negative Werte möglich sind.

Schwierigkeitsgrad: Fortgeschritten **Vorkenntnisse**: Grundlegende Kenntnisse der 4diac-IDE und des IEC 61131-3-Funktionsbausteinsystems, Verständnis von Adapter-Schnittstellen. **Lernziele**:

- Arbeiten mit dem Zähler-Baustein `ADI_FB_CTUD`
- Konfiguration von digitalen Ein-/Ausgängen über logiBUS-Adapter
- Umwandlung von Datentypen (DINT ⟶ Digitaleingang, DINT ⟶ LREAL) für die Terminalausgabe
- Erzeugung von Impulsen für das Laden des Zählwerts (PV)

Verwendete Funktionsbausteine (FBs)

Die Übung besteht aus einer flachen Netzwerkstruktur ohne weitere Sub-Applikationen. Folgende Funktionsbausteine kommen zum Einsatz:

- **`ADI_FB_CTUD`** (Typ: `adapter::iec61131::counters::ADI_FB_CTUD`) Der zentrale Vorwärts-/Rückwärtszähler. Er besitzt die Adapter-Schnittstelle `CU` (Count Up), `CD` (Count Down), `R` (Reset), `LD` (Load), `PV` (Preset Value) sowie die Ausgänge `QU` (Überlauf), `QD` (Unterlauf) und `CV` (aktueller Zählerwert).
- **`ADI_DINT_TO_DI`** (Typ: `adapter::conversion::unidirectional::ADI_DINT_TO_DI`) Wandelt einen DINT-Wert in ein digitales Signal (Adapter-Schnittstelle) um. Der Parameter `OUT` ist auf `DINT#5` gesetzt, d. h. der Preset-Wert für den Zähler wird auf 5 voreingestellt.

- **Input_CU**, **Input_CD**, **Input_R**, **Input_LD** (Typ: logiBUS::io::DI::logiBUS_IXA) Digitale Eingangsadapter für die logiBUS-Hardware. Sie lesen die physischen Eingänge I1, I2, I3 und I4. Der Parameter QI ist auf TRUE gesetzt.
- **Output_QU**, **Output_QD** (Typ: logiBUS::io::DQ::logiBUS_QXA) Digitale Ausgangsadapter. Output_QU schaltet den physischen Ausgang Q1, Output_QD den Ausgang Q2. Beide haben QI = TRUE.
- **ADI_TO_ALR** (Typ: adapter::conversion::unidirectional::ADI_TO_ALR) Wandelt den Adapter-Ausgang CV (Zählerwert) in den Datentyp ALR (Analog-LREAL-Darstellung) um.
- **Q_NumericValue_PHYSA_LREAL** (Typ: isobus::UT::Q::Q_NumericValue_PHYSA_LREAL) Gibt den numerischen Wert (LREAL) auf ein Terminal aus. Der Parameter stObj verweist auf das konstante Objekt OutputNumber_N3 aus der Bibliothek Uebungen::const::UT::DefaultPool_Numeric.

Parameterdetails ausgewählter Bausteine

Baustein	Parameter	Wert
ADI_DINT_TO_DI	OUT	DINT#5
Input_CU	QI	TRUE
	Input	Input_I1
Input_CD	QI	TRUE
	Input	Input_I2
Input_R	QI	TRUE
	Input	Input_I3
Input_LD	QI	TRUE
	Input	Input_I4
Output_QU	QI	TRUE
	Output	Output_Q1
Output_QD	QI	TRUE
	Output	Output_Q2
Q_NumericValue_PHYSA_LREAL	stObj	OutputNumber_N3

Programmablauf und Verbindungen

Signalfluss

1. **Eingänge**: Die vier digitalen Eingänge (I1–I4) werden über die logiBUS-Adapter Input_CU, Input_CD, Input_R, Input_LD in die Steuerung eingelesen.
2. **Zählersteuerung**:
   - CU (Count Up) von Input_CU: jedes Ereignis am Eingang I1 erhöht den Zähler um 1.
   - CD (Count Down) von Input_CD: Ereignis an I2 verringert den Zähler um 1.
   - R (Reset) von Input_R: Ereignis an I3 setzt den Zähler auf 0 zurück.
   - LD (Load) von Input_LD: Ereignis an I4 lädt den Preset-Wert (PV) in den Zähler.
3. **Preset-Wert (PV)**: Der Baustein ADI_DINT_TO_DI wird beim INIT-Ereignis von Input_LD aktiviert (Event-Verbindung Input_LD.INITO → ADI_DINT_TO_DI.REQ). Er gibt den konstanten Wert DINT#5 an den Adapter-Eingang PV des Zählers weiter. Somit wird bei jedem Ladevorgang der Zähler auf 5 gesetzt.
4. **Ausgänge**:
   - QU (Count Up Overflow): geht auf TRUE, wenn der Zähler seinen maximalen Wert erreicht oder überschreitet → wird auf Output_Q1 ausgegeben.
   - QD (Count Down Overflow): TRUE bei Unterschreitung des Minimalwerts → Output_Q2.
   - CV (Current Value) wird über ADI_TO_ALR in ein LREAL-Signal gewandelt und an Q_NumericValue_PHYSA_LREAL übergeben. Dieses gibt den aktuellen Zählerstand als numerischen Wert auf dem Terminal (physikalische Ausgabe) aus.

Hinweise zum Aufbau

- **Kommentare im Netzwerk**:

  „hier sind negative Werte möglich !“ – Der Zähler `ADI_FB_CTUD` arbeitet mit DINT, daher können negative Zählerstände auftreten (z. B. durch mehr Rückwärts-als Vorwärtsimpulse). *„hier gegebenenfalls je einen AX_D_FF einbauen, damit die Events reduziert werden.“* – Bei schnellen Impulsfolgen könnte es erforderlich sein, Flanken-Filter (z. B. `AX_D_FF`) zwischen Eingängen und Zähler zu schalten, um die Ereignisrate zu begrenzen und Zählfehler zu vermeiden.
- **Keine eigenen Sub-Applikationen**: Der gesamte Programmablauf ist in einer Ebene realisiert.
- Die Verbindungen sind als **Adapter-Connections** ausgeführt, d. h. die Daten- und Ereignisübertragung erfolgt über Adapter-Schnittstellen.
- Die **Ereignisverbindung** `Input_LD.INITO → ADI_DINT_TO_DI.REQ` stellt sicher, dass der Preset-Wert nur beim Start des Eingangsbausteins (Initialisierung) neu gesendet wird.

Starten der Übung

1. Die Übung ist als SubAppType (`Uebung_221b_ALR`) in der 4diac-IDE eingebunden.
2. Voraussetzung ist eine laufende logiBUS-Hardware mit angeschlossenen Ein-/Ausgängen (`I1`–`I4`, `Q1`, `Q2`).
3. Das Terminal-Objekt `OutputNumber_N3` muss im Projekt vorhanden sein (aus der Bibliothek `Uebungen::const::UT::DefaultPool_Numeric`).
4. Nach dem Deployment kann die Steuerung durch Anlegen von Impulsen an den Eingängen getestet werden.

Zusammenfassung

Die Übung `Uebung_221b_ALR` demonstriert den Einsatz eines industriellen Vorwärts-/Rückwärtszählers (`ADI_FB_CTUD`) in der 4diac-IDE. Durch die Kombination von logiBUS-Eingängen, Datenkonvertierung und Terminalausgabe wird ein vollständiger Signalpfad von der Hardware bis zur Visualisierung abgebildet. Der Zähler kann über vier digitale Eingänge gesteuert werden, wobei ein fester Preset-Wert von 5 verwendet wird. Die Ausgabe des aktuellen Zählerstands als Gleitkommazahl auf das Terminal erleichtert die Überwachung und Fehlersuche. Die Übung vermittelt praxisnahe Kenntnisse über Adapter-Schnittstellen, Ereignissteuerung und Datentypkonvertierung.

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

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

2.1.2.512 Uebung_222_ALI: Standard IEC 61131-3 ALI_FB_CTUD (Adapter Version, Vorwärts-/Rückwärtszähler, LINT) mit Terminal-Ausgabe

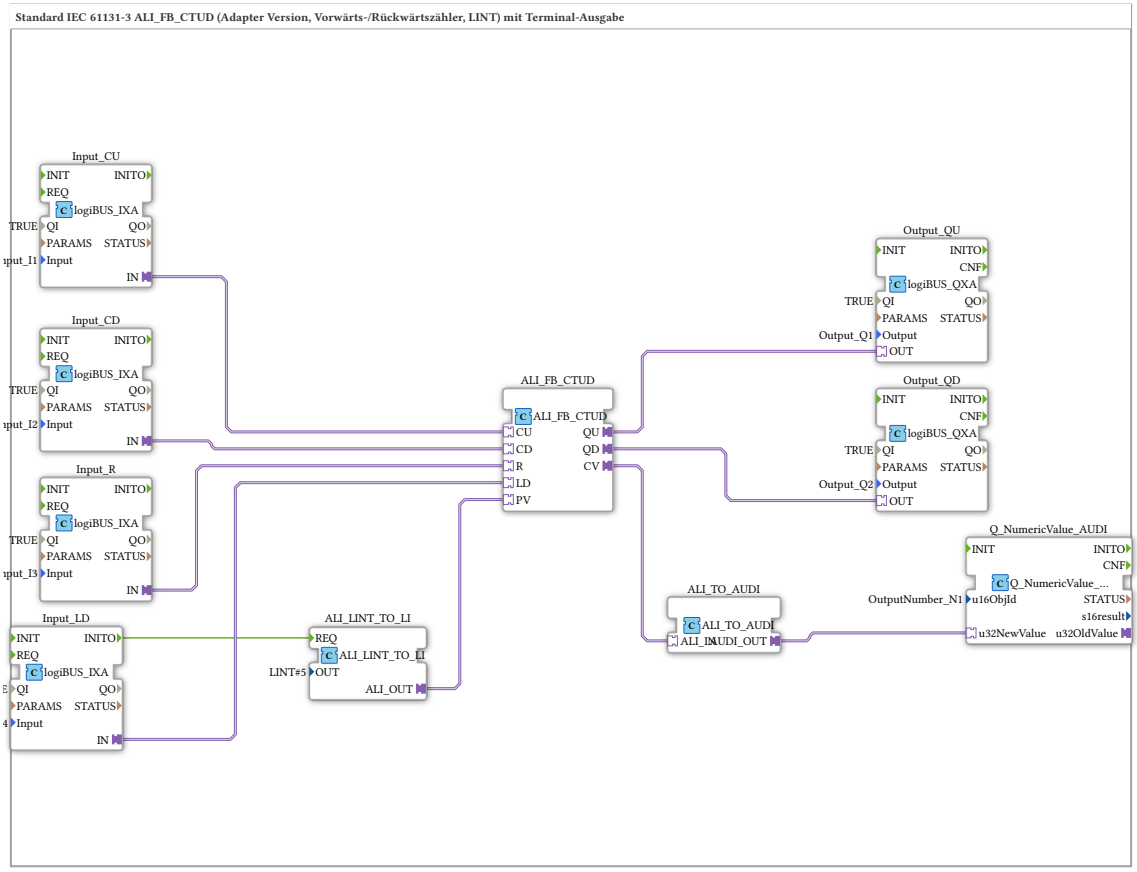


Bild nicht vorhanden

• – ○ • – ○ • – ○ •

Einleitung

Diese Übung implementiert einen Vorwärts-/Rückwärtszähler nach IEC 61131-3 (CTUD) im Adapter-Format. Der Zähler wird über digitale Eingänge (logiBUS) gesteuert und gibt seinen aktuellen Zählwert über ein Terminal aus. Der Preset-Wert (PV) wird auf LINT#5 gesetzt und beim Setzen des LD-Eingangs geladen. Die Ausgänge QU (Überlauf) und QD (Unterlauf) werden auf digitale Ausgänge geschaltet.

Verwendete Funktionsbausteine (FBs)

- **ALI_FB_CTUD** (Typ: `adapter::iec61131::counters::ALI_FB_CTUD`):
  - Parameter: keine im XML gesetzt (Standardwerte)
  - Ereigniseingänge: CU, CD, R, LD
  - Daten-Eingänge: PV (von ALI_LINT_TO_LI)
  - Ereignisausgänge: (keine sichtbar)
  - Datenausgänge: QU, QD, CV
- **ALI_LINT_TO_LI** (Typ: `adapter::conversion::unidirectional::ALI_LINT_TO_LI`):
  - Parameter: `OUT` = `LINT#5`
  - Ereigniseingang: REQ (verbunden mit Input_LD.INITO)
  - Datenausgang: ALI_OUT (verbunden mit ALI_FB_CTUD.PV)
- **Input_CU** (Typ: `logiBUS::io::DI::logiBUS_IXA`):
  - Parameter: `QI` = `TRUE`, `Input` = `Input_I1`
  - Adapterausgang: IN
- **Input_CD** (Typ: `logiBUS::io::DI::logiBUS_IXA`):
  - Parameter: `QI` = `TRUE`, `Input` = `Input_I2`
  - Adapterausgang: IN
- **Input_R** (Typ: `logiBUS::io::DI::logiBUS_IXA`):

  - Parameter: `QI` = `TRUE`, `Input` = `Input_I3`
  - Adapterausgang: IN
- **Input_LD** (Typ: `logiBUS::io::DI::logiBUS_IXA`):
  - Parameter: `QI` = `TRUE`, `Input` = `Input_I4`
  - Adapterausgang: IN
- **Output_QU** (Typ: `logiBUS::io::DQ::logiBUS_QXA`):
  - Parameter: `QI` = `TRUE`, `Output` = `Output_Q1`
  - Adaptereingang: OUT
- **Output_QD** (Typ: `logiBUS::io::DQ::logiBUS_QXA`):
  - Parameter: `QI` = `TRUE`, `Output` = `Output_Q2`
  - Adaptereingang: OUT
- **ALI_TO_AUDI** (Typ: `adapter::conversion::unidirectional::ALI_TO_AUDI`):
  - Konvertiert einen `ALI`-Wert in ein `AUDI`-Format (für Terminalausgabe)
  - **Hinweis:** Gemäß Kommentar ist dieser Baustein nicht geeignet, um negative Zahlen darzustellen.
- **Q_NumericValue_AUDI** (Typ: `isobus::UT::Q::Q_NumericValue_AUDI`):
  - Parameter: `u16ObjId` = `OutputNumber_N1`
  - Zeigt den übergebenen numerischen Wert auf einem Terminal an.

Programmablauf und Verbindungen

Die Hardware-Eingänge (I1–I4) werden über die logiBUS-DI-Bausteine eingelesen. Die Ereignisse und Daten werden wie folgt verbunden:

1. **Takteingänge CU und CD**:
   - `Input_CU.IN` ⟶ `ALI_FB_CTUD.CU` (Vorwärtszählen, positive Flanke)
   - `Input_CD.IN` ⟶ `ALI_FB_CTUD.CD` (Rückwärtszählen, positive Flanke)
2. **Rücksetzen und Laden**:
   - `Input_R.IN` ⟶ `ALI_FB_CTUD.R` (Setzt den Zählerstand auf 0)
   - `Input_LD.IN` ⟶ `ALI_FB_CTUD.LD` (Lädt den Preset-Wert PV in den Zähler)
   - Das Ereignis `Input_LD.INITO` löst den `ALI_LINT_TO_LI.REQ` aus, sodass der feste Wert LINT#5 als PV am Ausgang `ALI_OUT` bereitsteht.
3. **Preset-Wert**:
   - `ALI_LINT_TO_LI.ALI_OUT` ⟶ `ALI_FB_CTUD.PV`
4. **Ausgänge**:
   - `ALI_FB_CTUD.QU` ⟶ `Output_QU.OUT` (wird auf digitalen Ausgang Q1 geschaltet)
   - `ALI_FB_CTUD.QD` ⟶ `Output_QD.OUT` (wird auf digitalen Ausgang Q2 geschaltet)
5. **Zählwertausgabe auf Terminal**:
   - `ALI_FB_CTUD.CV` ⟶ `ALI_TO_AUDI.ALI_IN`
   - `ALI_TO_AUDI.AUDI_OUT` ⟶ `Q_NumericValue_AUDI.u32NewValue`
   - Der aktuelle Zählwert (CV) wird umgewandelt und im Terminal mit der Objekt-ID `OutputNumber_N1` ausgegeben.

Hinweise aus den Kommentaren:

- Der Baustein `ALI_TO_AUDI` unterstützt keine negativen Zahlen – der ausgegebene Wert kann daher bei Unterschreiten von 0 nicht korrekt dargestellt werden.
- Zur Reduzierung der Eventanzahl (z. B. bei schnellen Zählimpulsen) könnte man vor den Ausgängen jeweils einen `AX_D_FF` einfügen.

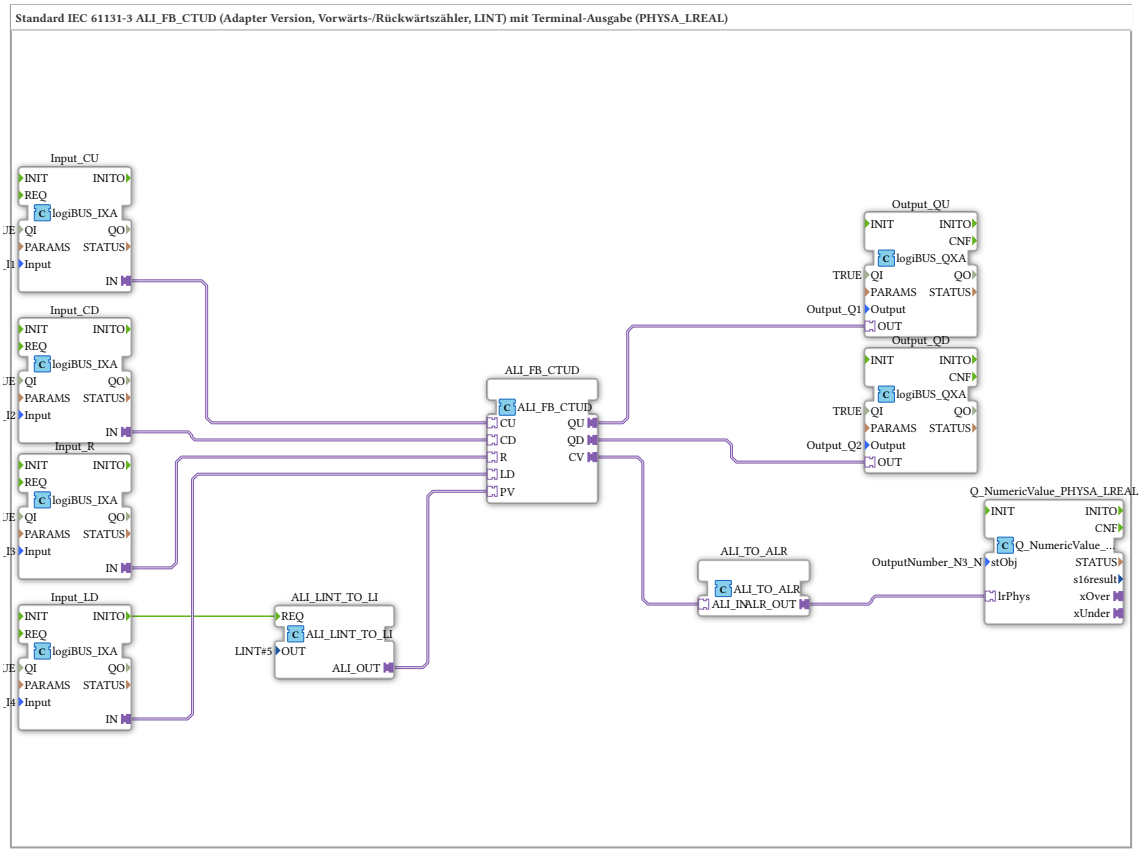
Zusammenfassung

Die Übung demonstriert den Einsatz des standardisierten CTUD-Zählers (IEC 61131-3) in einer 4diac-Umgebung unter Verwendung des Adapter-Konzepts. Die Anbindung an die reale Hardware (logiBUS) sowie die Ausgabe eines Zählwertes auf ein Terminal werden gezeigt. Der Preset-Wert wird einmalig beim Start durch einen eigenen Konvertierungsbaustein gesetzt. Die Übung macht auch auf Einschränkungen der Konvertierungsbausteine aufmerksam (keine negativen Zahlen) und gibt einen Hinweis zur Optimierung der Eventverarbeitung bei schnellen Signalen.

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

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

2.1.2.513 Uebung_222b_ALR: Standard IEC 61131-3 ALI_FB_CTUD (Adapter Version, Vorwärts-/Rückwärtszähler, LINT) mit Terminal-Ausgabe (PHYSA_LREAL)



• — ○ • — ○ • — ○ •

Einleitung

Diese Übung implementiert einen Vorwärts-/Rückwärtszähler auf Basis des adaptierten IEC-61131-3-Bausteins `ALI_FB_CTUD` (Typ LINT). Die Zählerimpulse, der Reset und der Ladevorgang werden über vier digitale Eingänge (logiBUS) gesteuert. Die beiden Zählausgänge `QU` und `QD` werden auf digitale Ausgänge geführt. Der aktuelle Zählerstand (`CV`) wird über einen Konverter in einen physikalischen Wert (LREAL) umgewandelt und auf einem Terminal (Ausgabepool) ausgegeben. Ein konstanter Startwert (`PV = 5`) wird über einen separaten Konvertierungsbaustein bereitgestellt.

Lernziele:

- Verwendung des adaptierten IEC-Zählers `ALI_FB_CTUD`
- Verbindung von digitalen Ein-/Ausgängen (logiBUS)
- Konvertierung zwischen LINT und LREAL
- Ausgabe eines physikalischen Werts auf ein Terminal (Pool-Element)

Verwendete Funktionsbausteine (FBs)

Die Applikation besteht aus folgenden Bausteinen, die im SubApp-Netzwerk verbunden sind:

- **ALI_FB_CTUD** (`adapter::iec61131::counters::ALI_FB_CTUD`) Der zentrale Vorwärts-/Rückwärtszähler (IEC 61131-3, LINT). Er besitzt die Adapter-Eingänge `CU`, `CD`, `R`, `LD`, die Ausgänge `QU`, `QD` sowie den Datenausgang `CV`.
- **ALI_LINT_TO_LI** (`adapter::conversion::unidirectional::ALI_LINT_TO_LI`) Wandelt einen konstanten LINT-Wert in ein LINT-Signal um. Parameter: `OUT = LINT#5` (Startwert für den Zähler).

- **Input_CU** (`logiBUS::io::DI::logiBUS_IXA`) Digitaler Eingang für den Vorwärtszählimpuls (`CU`), verbunden mit `Input_I1`. Parameter: `QI = TRUE`.
- **Input_CD** (`logiBUS::io::DI::logiBUS_IXA`) Digitaler Eingang für den Rückwärtszählimpuls (`CD`), verbunden mit `Input_I2`. Parameter: `QI = TRUE`.
- **Input_R** (`logiBUS::io::DI::logiBUS_IXA`) Digitaler Eingang für den Reset (`R`), verbunden mit `Input_I3`. Parameter: `QI = TRUE`.
- **Input_LD** (`logiBUS::io::DI::logiBUS_IXA`) Digitaler Eingang für das Laden des Startwerts (`LD`), verbunden mit `Input_I4`. Parameter: `QI = TRUE`.
- **Output_QU** (`logiBUS::io::DQ::logiBUS_QXA`) Digitaler Ausgang für den Zählausgang `QU`, verbunden mit `Output_Q1`. Parameter: `QI = TRUE`.
- **Output_QD** (`logiBUS::io::DQ::logiBUS_QXA`) Digitaler Ausgang für den Zählausgang `QD`, verbunden mit `Output_Q2`. Parameter: `QI = TRUE`.
- **ALI_TO_ALR** (`adapter::conversion::unidirectional::ALI_TO_ALR`) Wandelt den LINT-Zählerstand (`CV`) in einen LREAL-Wert (physikalische Größe) um.
- **Q_NumericValue_PHYSA_LREAL** (`isobus::UT::Q::Q_NumericValue_PHYSA_LREAL`) Nimmt den physikalischen Wert auf und gibt ihn auf das konfigurierte Terminalelement (`OutputNumber_N3`) aus.

Programmablauf und Verbindungen

Die logischen Verbindungen (über Adapter) stellen den Datenfluss her:

1. **Eingänge zum Zähler**
   - `Input_CU.IN` ⟶ `ALI_FB_CTUD.CU`
   - `Input_CD.IN` ⟶ `ALI_FB_CTUD.CD`
   - `Input_R.IN` ⟶ `ALI_FB_CTUD.R`
   - `Input_LD.IN` ⟶ `ALI_FB_CTUD.LD`
2. **Startwert (PV)**
   - `Input_LD.INITO` (Ereignis) ⟶ `ALI_LINT_TO_LI.REQ`
   - `ALI_LINT_TO_LI.ALI_OUT` ⟶ `ALI_FB_CTUD.PV` Der Startwert wird beim Laden (Flanke an `LD`) aktiv gesetzt.
3. **Zählerausgänge**
   - `ALI_FB_CTUD.QU` ⟶ `Output_QU.OUT`
   - `ALI_FB_CTUD.QD` ⟶ `Output_QD.OUT`
4. **Ausgabe des Zählerstands (Terminal)**
   - `ALI_FB_CTUD.CV` ⟶ `ALI_TO_ALR.ALI_IN`
   - `ALI_TO_ALR.ALR_OUT` ⟶ `Q_NumericValue_PHYSA_LREAL.lrPhys` Der Zählerwert (LINT) wird in einen LREAL-Wert umgewandelt und an die Terminal-Ausgabe gesendet.

Beachten Sie die Kommentare im Netzwerk:

- *„hier sind negative Werte möglich!“* (der LINT-Zähler kann unter Null zählen)
- *„hier gegebenenfalls je einen AX_D_FF einbauen, damit die Events reduziert werden.“* (Hinweis zur Ereignisoptimierung an den Ausgängen)

Der Baustein `ALI_LINT_TO_LI` initialisiert den Vorgabewert mit `LINT#5`. Dadurch wird der Zähler beim ersten Ladevorgang auf 5 gesetzt.

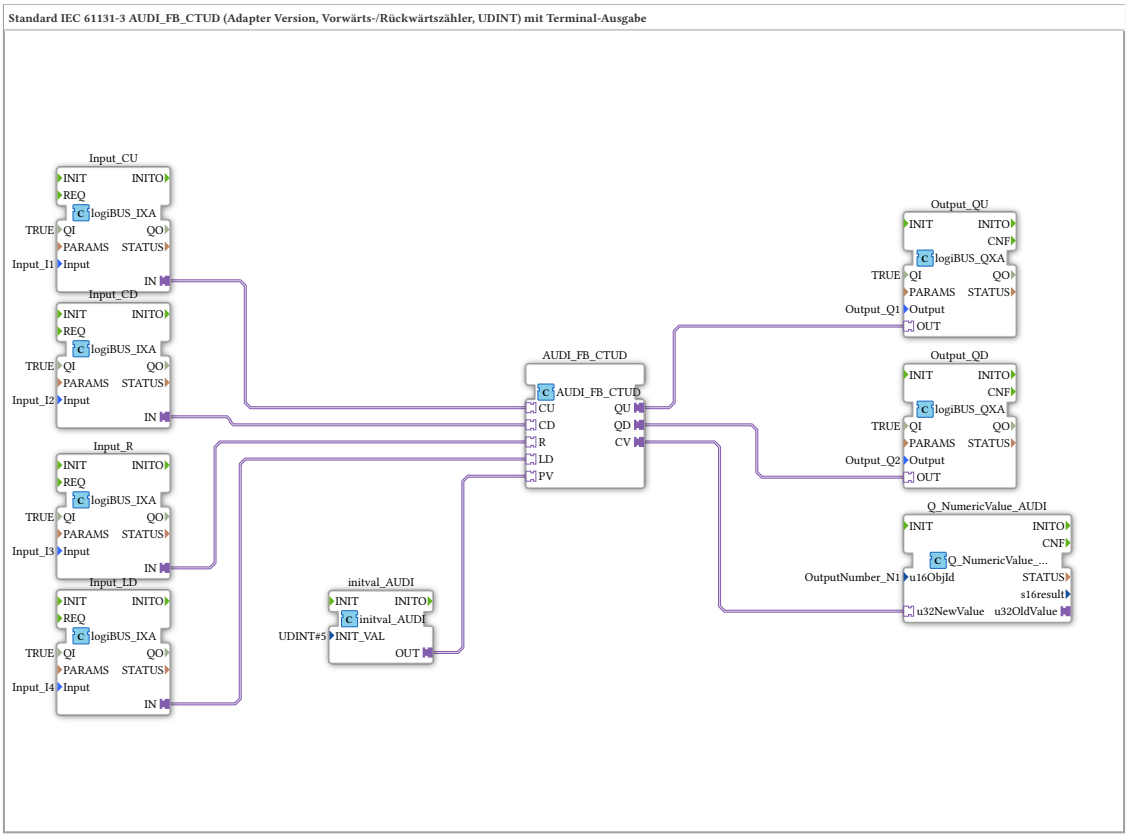
Zusammenfassung

Die Übung 222b demonstriert den Einsatz eines adaptierten IEC-Zählers in der 4diac-IDE. Über digitale logiBUS-Eingänge werden Zählimpulse, Reset und Laden gesteuert. Der aktuelle Zählerstand wird sowohl auf digitale Ausgänge als auch – nach Konvertierung in eine physikalische Größe – auf ein Terminal ausgegeben. Sie lernen die Verbindung von Ein-/Ausgabebausteinen, die Konvertierung von Ganzzahl- in Gleitkommawerte sowie das Setzen eines konstanten Startwerts kennen. Hinweise zu negativen Zählerständen und zur Ereignisreduzierung ergänzen die praktische Anwendung.

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

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

2.1.2.514 Uebung_223_AUDI: Standard IEC 61131-3 AUDI_FB_CTUD (Adapter-Version, Vorwärts-/Rückwärtszähler, UDINT) mit Terminal-Ausgabe



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung zeigt einen Vorwärts-/Rückwärtszähler (CTUD) nach IEC 61131-3 als Adapter-Version für den Datentyp UDINT. Vier digitale Eingänge steuern Aufwärtszählen, Abwärtszählen, Reset und Laden, der feste Vorgabewert `5` kommt aus einem `initval_AUDI`-Konstantenbaustein. Der Zählerstand wird auf dem ISOBUS-Terminal angezeigt, die Zählerausgänge `QU` und `QD` liegen an `Q1` und `Q2`.

Verwendete Funktionsbausteine (FBs)

- **Input_CU**: Eingang Vorwärtszählen
  - **Typ**: `logiBUS::io::DI::logiBUS_IXA`
  - **Parameter**: `QI = TRUE`, `Input = Input_I1`
  - **Erklärung**: Stellt den Zustand von `I1` als Adapterausgang `IN` bereit.
- **Input_CD**: Eingang Rückwärtszählen
  - **Typ**: `logiBUS::io::DI::logiBUS_IXA`
  - **Parameter**: `QI = TRUE`, `Input = Input_I2`
  - **Erklärung**: Stellt den Zustand von `I2` als Adapterausgang `IN` bereit.
- **Input_R**: Reset-Eingang
  - **Typ**: `logiBUS::io::DI::logiBUS_IXA`
  - **Parameter**: `QI = TRUE`, `Input = Input_I3`
  - **Erklärung**: Stellt den Zustand von `I3` als Adapterausgang `IN` bereit.
- **Input_LD**: Eingang Laden
  - **Typ**: `logiBUS::io::DI::logiBUS_IXA`
  - **Parameter**: `QI = TRUE`, `Input = Input_I4`
  - **Erklärung**: Stellt den Zustand von `I4` als Adapterausgang `IN` bereit.

- **initval_AUDI**: Konstante für den Vorgabewert
  - **Typ**: `adapter::types::unidirectional::AUDI::initval::initval_AUDI`
  - **Parameter**: `INIT_VAL = 5`
  - **Erklärung**: Liefert den Vorgabewert `5` als `AUDI`-Adapterwert an `AUDI_FB_CTUD.PV`.
- **AUDI_FB_CTUD**: Vorwärts-/Rückwärtszähler (UDINT, Adapter-Version)
  - **Typ**: `adapter::iec61131::counters::AUDI_FB_CTUD`
  - **Parameter**: keine
  - **Erklärung**: Zählt über `CU` aufwärts und über `CD` abwärts, `R` setzt zurück, `LD` lädt den Vorgabewert `PV`. `QU` und `QD` sind die Zählerausgänge nach IEC 61131-3 (CTUD), `CV` liefert den aktuellen Zählerstand.
- **Output_QU**: Digitalausgang für `QU`
  - **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
  - **Parameter**: `QI = TRUE`, `Output = Output_Q1`
  - **Erklärung**: Gibt `AUDI_FB_CTUD.QU` am physischen Ausgang `Q1` aus.
- **Output_QD**: Digitalausgang für `QD`
  - **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
  - **Parameter**: `QI = TRUE`, `Output = Output_Q2`
  - **Erklärung**: Gibt `AUDI_FB_CTUD.QD` am physischen Ausgang `Q2` aus.
- **Q_NumericValue_AUDI**: Anzeige auf dem ISOBUS-Terminal
  - **Typ**: `isobus::UT::Q::Q_NumericValue_AUDI`
  - **Parameter**: `u16ObjId = OutputNumber_N1`
  - **Erklärung**: Empfängt den Zählerstand über den `AUDI`-Adapter `u32NewValue` und zeigt ihn im Objekt `OutputNumber_N1` an.

Programmablauf und Verbindungen

1. `Input_CU.IN` ⟶ `AUDI_FB_CTUD.CU`, `Input_CD.IN` ⟶ `AUDI_FB_CTUD.CD`, `Input_R.IN` ⟶ `AUDI_FB_CTUD.R` und `Input_LD.IN` ⟶ `AUDI_FB_CTUD.LD` (AdapterConnections): Die Zustände von `I1` bis `I4` steuern Zählen, Rückwärtszählen, Reset und Laden.
2. `initval_AUDI.OUT` ⟶ `AUDI_FB_CTUD.PV` (AdapterConnection): Der Vorgabewert `5` wird als Preset übergeben.
3. `AUDI_FB_CTUD.QU` ⟶ `Output_QU.OUT` und `AUDI_FB_CTUD.QD` ⟶ `Output_QD.OUT` (AdapterConnections): Die Zählerausgänge liegen an `Q1` und `Q2`.
4. `AUDI_FB_CTUD.CV` ⟶ `Q_NumericValue_AUDI.u32NewValue` (AdapterConnection): Der Zählerstand wird auf dem Terminal angezeigt.

Verhalten: `I1` zählt aufwärts, `I2` abwärts, `I3` setzt den Zähler zurück und `I4` lädt den Vorgabewert `5`. Der Zählerstand erscheint in `OutputNumber_N1`. Laut Netzwerk-Kommentar kann hier gegebenenfalls je ein `AX_D_FF` eingebaut werden, damit die Events reduziert werden.

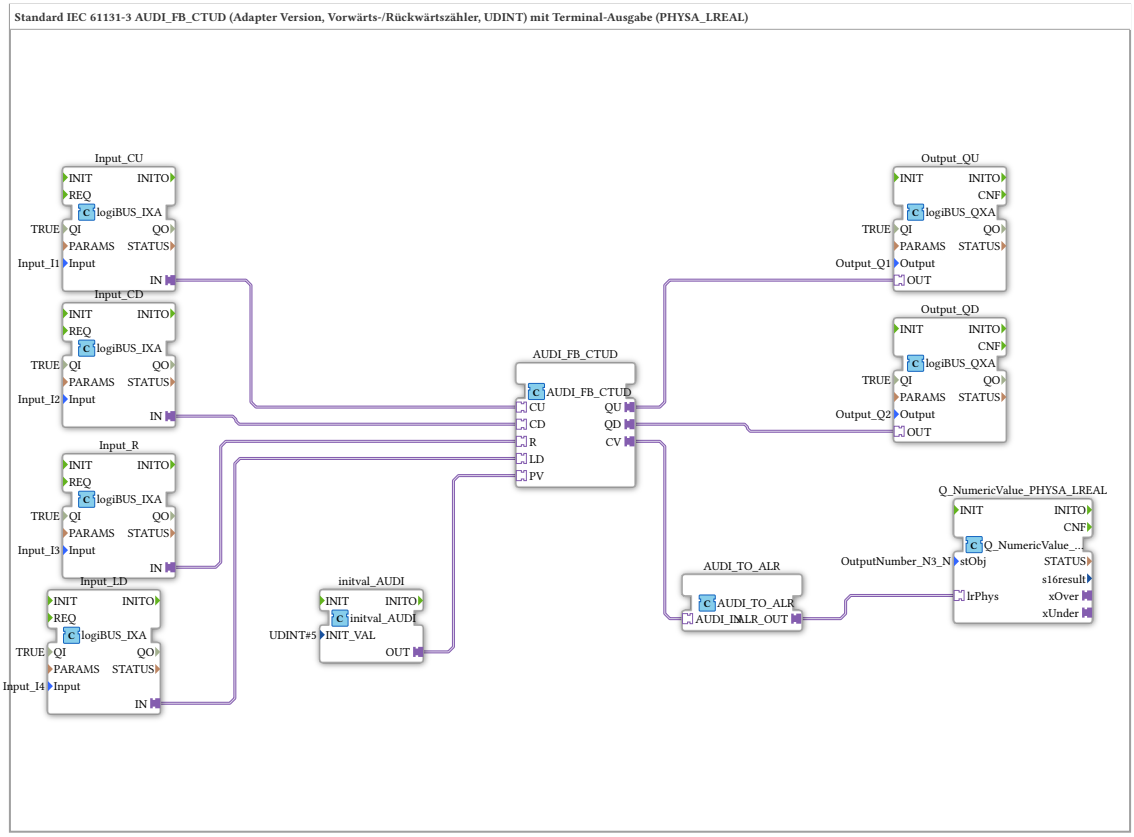
Zusammenfassung

Die Übung zeigt die adapterbasierte Anbindung eines IEC-61131-3-Auf-/Abwärtszählers: Eingänge, Vorgabewert, Zählerausgänge und Anzeige sind rein über Adapterverbindungen verschaltet.

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

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

2.1.2.515 Uebung_223b_ALR: Standard IEC 61131-3 AUDI_FB_CTUD (Adapter-Version, Vorwärts-/Rückwärtszähler, UDINT) mit Terminal-Ausgabe (PHYSA_LREAL)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung realisiert einen Vorwärts-/Rückwärtszähler (CTUD) nach IEC 61131-3 mit dem Zählbereich `UDINT` als Adapter-Version. Vier Taster steuern den Zähler: `I1` zählt hoch (`CU`), `I2` zählt runter (`CD`), `I3` setzt zurück (`R`), `I4` löst das Laden (`LD`) aus. Der Vorgabewert `PV` ist über `initval_AUDI` fest auf 5 gesetzt. Die Ausgänge `QU` und `QD` schalten `Q1` und `Q2`; der Zählerstand `CV` wird in einen `LREAL`-Wert gewandelt und am Terminal-Objekt `OutputNumber_N3_N` ausgegeben.

Verwendete Funktionsbausteine (FBs)

- **Input_CU**, **Input_CD**, **Input_R**, **Input_LD**: Digitaleingänge
  - **Typ**: `logiBUS::io::DI::logiBUS_IXA`
  - **Parameter**: `QI = TRUE`, `Input = Input_I1` (`Input_CU`), `Input_I2` (`Input_CD`), `Input_I3` (`Input_R`), `Input_I4` (`Input_LD`)
  - **Erklärung**: Stellen den Zustand des jeweiligen Eingangs als Adapter `IN` bereit; sie gehen an `CU`, `CD`, `R` bzw. `LD` des Zählers.
- **initval_AUDI**: Vorgabewert
  - **Typ**: `adapter::types::unidirectional::AUDI::initval::initval_AUDI`
  - **Parameter**: `INIT_VAL = 5`
  - **Erklärung**: Liefert den Wert 5 als Vorgabewert `PV` an den Zähler.
- **AUDI_FB_CTUD**: Vorwärts-/Rückwärtszähler
  - **Typ**: `adapter::iec61131::counters::AUDI_FB_CTUD`
  - **Parameter**: keine
  - **Erklärung**: Eingänge `CU`, `CD`, `R`, `LD` und `PV`; Ausgänge `QU`, `QD` und `CV` (aktueller Zählerstand), jeweils als Adapter.
- **Output_QU**, **Output_QD**: logiBUS Digitalausgänge
  - **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
  - **Parameter**: `QI = TRUE`, `Output = Output_Q1` (`Output_QU`), `Output_Q2` (`Output_QD`)

  - **Erklärung**: Geben `AUDI_FB_CTUD.QU` an `Q1` und `AUDI_FB_CTUD.QD` an `Q2` aus.
- **AUDI_TO_ALR**: Konvertierung `AUDI` nach `ALR`
  - **Typ**: `adapter::conversion::unidirectional::AUDI_TO_ALR`
  - **Parameter**: keine
  - **Erklärung**: Wandelt den Zählerstand (`AUDI_IN`) in einen `LREAL`-Adapterwert (`ALR_OUT`). Laut Kommentar im Netzwerk sind hier negative Werte möglich.
- **Q_NumericValue_PHYSA_LREAL**: Terminal-Ausgabe
  - **Typ**: `isobus::UT::Q::Q_NumericValue_PHYSA_LREAL`
  - **Parameter**: `stObj = OutputNumber_N3_N`
  - **Erklärung**: Zeigt den `LREAL`-Wert aus `lrPhys` am Terminal-Objekt `OutputNumber_N3_N` an.

Programmablauf und Verbindungen

1. `Input_CU.IN` ⟶ `AUDI_FB_CTUD.CU`, `Input_CD.IN` ⟶ `AUDI_FB_CTUD.CD`, `Input_R.IN` ⟶ `AUDI_FB_CTUD.R` und `Input_LD.IN` ⟶ `AUDI_FB_CTUD.LD` (AdapterConnections): Die vier Taster steuern den Zähler.
2. `initval_AUDI.OUT` ⟶ `AUDI_FB_CTUD.PV` (AdapterConnection): Vorgabewert 5.
3. `AUDI_FB_CTUD.QU` ⟶ `Output_QU.OUT` und `AUDI_FB_CTUD.QD` ⟶ `Output_QD.OUT` (AdapterConnections): Die Zählerausgänge schalten `Q1` und `Q2`.
4. `AUDI_FB_CTUD.CV` ⟶ `AUDI_TO_ALR.AUDI_IN` und `AUDI_TO_ALR.ALR_OUT` ⟶ `Q_NumericValue_PHYSA_LREAL.lrPhys` (AdapterConnections): Der Zählerstand wird als `LREAL` am Terminal angezeigt.

Hinweis aus dem Netzwerk: Hier kann gegebenenfalls je ein `AX_D_FF` eingebaut werden, damit die Events reduziert werden.

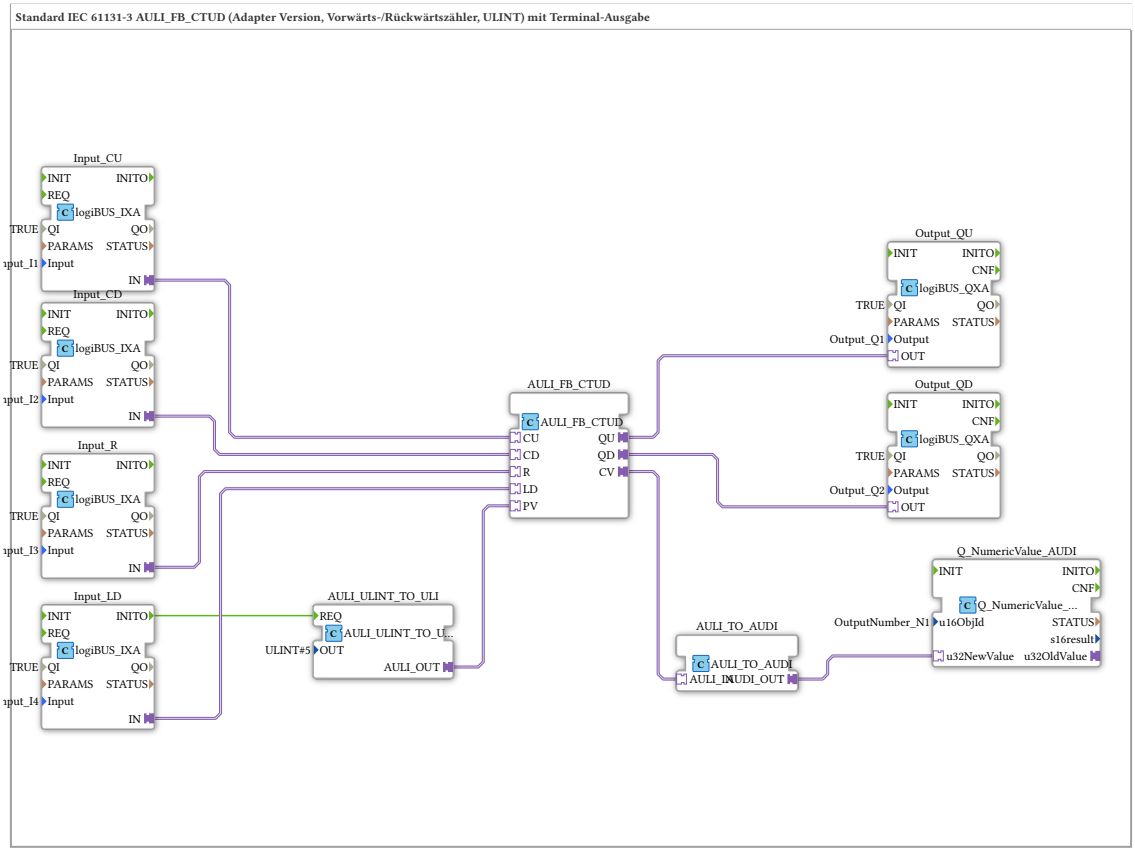
Zusammenfassung

Die Übung 223b_ALR zeigt einen adapterbasierten IEC-Vorwärts-/Rückwärtszähler mit festem Vorgabewert 5 und Terminal-Ausgabe des Zählerstands über `AUDI_TO_ALR`. Alle Verbindungen sind Adapterverbindungen.

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

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

2.1.2.516 Uebung_224_AULI: Standard IEC 61131-3 AULI_FB_CTUD (Adapter Version, Vorwärts-/Rückwärtszähler, ULINT) mit Terminal-Ausgabe



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung realisiert einen Vorwärts-/Rückwärtszähler gemäß IEC 61131-3 (Typ CTUD) im Adapter-Format. Der Zähler verwendet den Datentyp ULINT (Unsigned Long Integer) und gibt den aktuellen Zählerstand sowie die Überlauf-/Unterlauf-Signale an physische Ausgänge aus. Zusätzlich wird der Zählerstand über ein Terminal (isobus) ausgegeben. Der Preset-Wert (PV) wird initial auf 5 gesetzt.

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: `AULI_FB_CTUD`

- **Typ**: `adapter::iec61131::counters::AULI_FB_CTUD`
- **Verwendete interne FBs**: keine (Funktionsbaustein direkt verwendet)
- **Funktionsweise**: IEC 61131-3 CTUD-Zähler mit den Anschlüssen:
  - **CU** (Count Up) – Vorwärtszählen bei positiver Flanke
  - **CD** (Count Down) – Rückwärtszählen bei positiver Flanke
  - **R** (Reset) – Setzt Zählerstand zurück auf 0
  - **LD** (Load) – Lädt den Preset-Wert (PV) in den Zähler
  - **PV** (Preset Value) – Vorgabewert (ULINT, initial auf 5 gesetzt)
  - **QU** (Overflow) – Ausgang wird TRUE, wenn Zählerstand ≥ PV (bei Aufwärtszählen)
  - **QD** (Underflow) – Ausgang wird TRUE, wenn Zählerstand ≤ 0 (bei Abwärtszählen)
  - **CV** (Current Value) – aktueller Zählerstand (ULINT)

Sub-Bausteine: `AULI_ULINT_TO_ULI`

- **Typ**: `adapter::conversion::unidirectional::AULI_ULINT_TO_ULI`
- **Verwendete interne FBs**: keine
  - **Parameter**: `OUT = ULINT#5`

- **Funktionsweise**: Wandelt einen konstanten ULINT-Wert (5) in ein ULI-Format um, das als Preset-Wert (PV) für den Zähler verwendet wird. Der Baustein wird beim Start der Applikation einmalig ausgelöst (über Eventverbindung `Input_LD.INIT0 → AULI_ULINT_TO_ULI.REQ`).

Sub-Bausteine: `AULI_TO_AUDI`

- **Typ**: `adapter::conversion::unidirectional::AULI_TO_AUDI`
- **Verwendete interne FBs**: keine
- **Funktionsweise**: Konvertiert den aktuellen Zählerstand (AULI-Format) in ein AUDI-Format, das für die Terminalausgabe benötigt wird.

Sub-Bausteine: `Q_NumericValue_AUDI`

- **Typ**: `isobus::UT::Q::Q_NumericValue_AUDI`
- **Verwendete interne FBs**: keine
  - **Parameter**: `u16ObjId = OutputNumber_N1` (Referenz auf Terminalausgabeobjekt)
- **Funktionsweise**: Nimmt den konvertierten Zählerstand (AUDI) entgegen und gibt ihn als numerischen Wert an das konfigurierte Terminal (isobus) aus.

Weitere verwendete FBs (logiBUS E/A-Anbindung)

- **Input_CU** (`logiBUS::io::DI::logiBUS_IXA`): Liest den digitalen Eingang `Input_I1` (Vorwärtszählen)
- **Input_CD** (`logiBUS::io::DI::logiBUS_IXA`): Liest den digitalen Eingang `Input_I2` (Rückwärtszählen)
- **Input_R** (`logiBUS::io::DI::logiBUS_IXA`): Liest den digitalen Eingang `Input_I3` (Reset)
- **Input_LD** (`logiBUS::io::DI::logiBUS_IXA`): Liest den digitalen Eingang `Input_I4` (Load)
- **Output_QU** (`logiBUS::io::DQ::logiBUS_QXA`): Schaltet den digitalen Ausgang `Output_Q1` (Überlauf)
- **Output_QD** (`logiBUS::io::DQ::logiBUS_QXA`): Schaltet den digitalen Ausgang `Output_Q2` (Unterlauf)

Alle logiBUS-Blöcke sind standardmäßig mit `QI = TRUE` aktiv geschaltet.

Programmablauf und Verbindungen

1. **Initialisierung**: Beim Systemstart löst der Baustein `Input_LD` über seinen Ereignisausgang `INIT0` den Baustein `AULI_ULINT_TO_ULI` aus. Dieser setzt den Preset-Wert des Zählers auf `ULINT#5`.
2. **Zählbetrieb**: Die vier digitalen Eingänge (`Input_I1` bis `Input_I4`) werden über Adapterverbindungen direkt mit den Zählereingängen `CU`, `CD`, `R` und `LD` des `AULI_FB_CTUD` verbunden. Der Zähler reagiert auf positive Flanken an den jeweiligen Eingängen.
3. **Ausgabe der Zählerstände**:
   - Die Überlauf- und Unterlauf-Signale `QU` und `QD` werden über die Adapterausgänge an die physischen Ausgänge `Output_Q1` und `Output_Q2` weitergeleitet.
   - Der aktuelle Zählerstand `CV` wird über die Konvertierungsbausteine `AULI_TO_AUDI` und `Q_NumericValue_AUDI` auf dem Terminal (Objekt `OutputNumber_N1`) dargestellt. Diese Konvertierung und Ausgabe erfolgt ebenfalls zyklisch parallel zum Zählbetrieb.

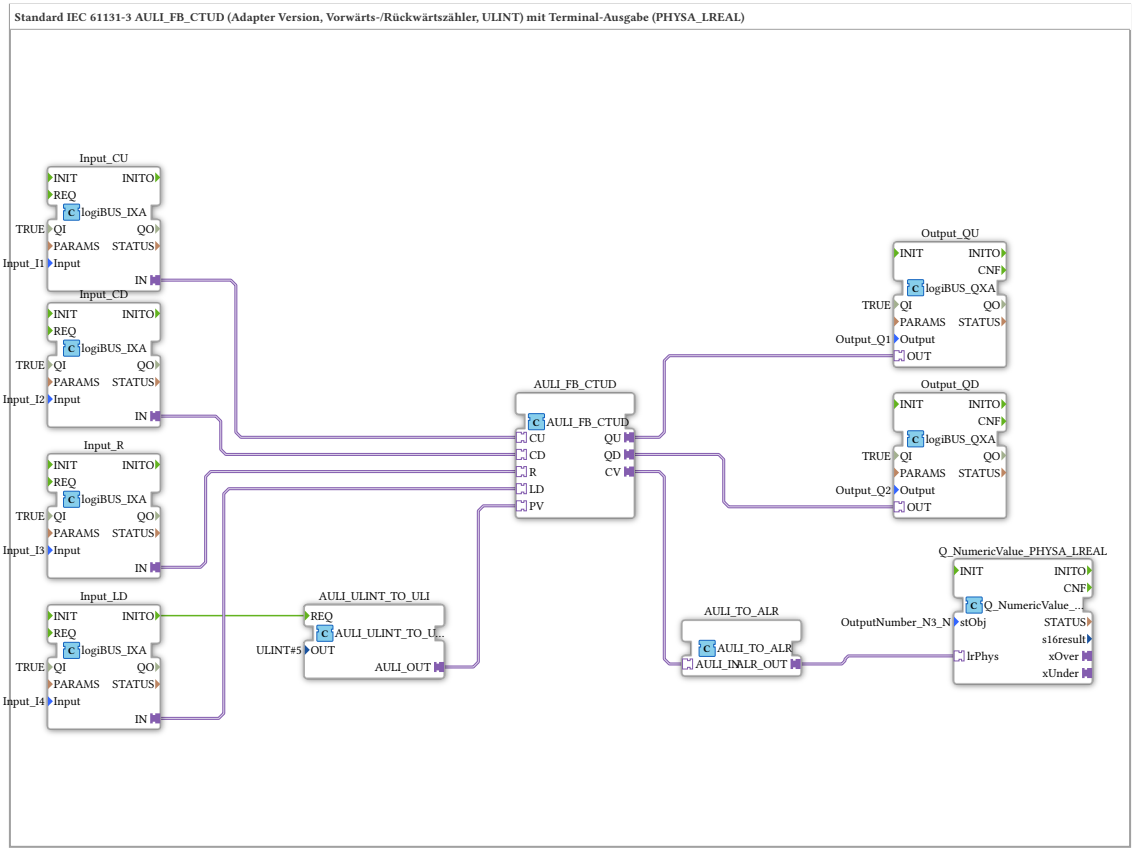
Hinweise:

- Ein Kommentar im Netzwerk schlägt vor, optional AX_D_FF (D-Flipflops) einzubauen, um die Ereignisrate zu reduzieren.
- Der Zähler arbeitet mit dem Datentyp ULINT, daher sind große Zählbereiche möglich.

Zusammenfassung

Die Übung demonstriert die Verwendung eines standardkonformen IEC 61131-3 CTUD-Zählers in der Adapterversion. Der Zähler wird über vier digitale Eingänge gesteuert, gibt Überlauf/Unterlauf auf Ausgänge aus und zeigt den aktuellen Stand auf einem Terminal an. Der Preset-Wert wird initial fest vorgegeben. Die Realisierung verwendet logiBUS-Ein-/Ausgänge und spezielle Konvertierungsbausteine, um den Zählerstand an das Terminal zu übergeben. Dieses Beispiel eignet sich als Grundlage für Zähleraufgaben in der Automatisierungstechnik.

2.1.2.517 Uebung_224b_ALR: Standard IEC 61131-3 AULI_FB_CTUD (Adapter Version, Vorwärts-/Rückwärtszähler, ULINT) mit Terminal-Ausgabe (PHYSA_LREAL)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung implementiert einen Standard IEC 61131-3 Vorwärts-/Rückwärtszähler (CTUD) als Adapter-Version mit dem Datentyp ULINT. Der aktuelle Zählerstand wird über einen Terminal-Ausgang in Form des physischen Wertes (PHYSA_LREAL) ausgegeben. Die Steuerung erfolgt über vier digitale Eingänge, zwei digitale Ausgänge sowie Startwertvorgabe über einen ULINT-zu-ULI-Konverter.

Verwendete Funktionsbausteine (FBs)

- **AULI_FB_CTUD**: Adapter-basierter Vorwärts-/Rückwärtszähler (Typ: ULINT)
  - **Parameter**: keine statischen Parameter
  - **Erklärung**: Der eigentliche Zähler. Er besitzt die Ereignis-/Adaptereingänge CU (Vorwärtszählen), CD (Rückwärtszählen), R (Reset), LD (Laden des Startwerts) und die Ausgänge QU (Überlauf?), QD (Unterlauf?), CV (aktueller Zählerstand). Über den Adaptereingang PV wird der Startwert geladen.
- **AULI_ULINT_TO_ULI**: Konvertierung von ULINT- nach ULI-Datentyp
  - **Parameter**: OUT = ULINT#5 (fester Startwert 5)
  - **Funktion**: Stellt den Startwert (PV) für den Zähler bereit.
- **Input_CU**, **Input_CD**, **Input_R**, **Input_LD**: logiBUS Digitaleingänge (Typ: logiBUS_IXA)
  - **Parameter**: QI = TRUE, Input = entsprechender physischer Eingang (Input_I1..I4)
  - **Funktion**: Wandeln die binären Eingangssignale (Taster, Schalter) in Adaptersignale für den Zähler um.
- **Output_QU**, **Output_QD**: logiBUS Digitalausgänge (Typ: logiBUS_QXA)
  - **Parameter**: QI = TRUE, Output = entsprechender physischer Ausgang (Output_Q1, Q2)
  - **Funktion**: Geben die Zählerausgänge QU (z.B. Überlauf) und QD (z.B. Unterlauf) an die Peripherie weiter.
- **AULI_TO_ALR**: Konvertierung von AULI (analoger Wert) nach ALR (LREAL)
  - **Parameter**: keine

  - **Funktion**: Wandelt den aktuellen Zählerstand (CV) vom Typ ULINT in den physischen Wert (LREAL) um.
- **Q_NumericValue_PHYSA_LREAL**: Terminal-Ausgabebaustein (Typ: iso-bus::UT::Q::Q_NumericValue_PHYSA_LREAL)
  - **Parameter**: stObj = OutputNumber_N3 (Referenz auf ein Terminal-Objekt)
  - **Funktion**: Gibt den konvertierten Wert auf dem Terminal aus.

Sub-Bausteine: keine

Die Übung verwendet keine weiteren Unterbausteine, alle FBs sind direkt auf der obersten Ebene der SubApp angeordnet.

Programmablauf und Verbindungen

1. **Initialisierung**: Beim Start (Ereignis INITO von Input_LD) wird der Baustein AULI_ULINT_TO_ULI getriggert, der den festen Startwert ULINT#5 an den PV-Eingang des Zählers weitergibt. Dadurch wird der Zähler auf 5 voreingestellt.
2. **Zählbetrieb**:
   - **Vorwärtszählen**: Impuls an Eingang I1 ⟶ Input_CU ⟶ Adapter CU ⟶ Zähler erhöht CV um 1.
   - **Rückwärtszählen**: Impuls an Eingang I2 ⟶ Input_CD ⟶ Adapter CD ⟶ Zähler verringert CV um 1 (negative Werte möglich!).
   - **Reset**: Impuls an Eingang I3 ⟶ Input_R ⟶ Adapter R ⟶ Zähler wird auf 0 zurückgesetzt.
   - **Laden**: Impuls an Eingang I4 ⟶ Input_LD ⟶ Adapter LD ⟶ Zähler lädt den Wert von PV (aktuell 5) in CV.
3. **Ausgabe**:
   - Bei Überlauf (QU) signalisiert Output_Q1.
   - Bei Unterlauf (QD) signalisiert Output_Q2.
   - Der aktuelle Zählerstand CV wird über die Konverterkette (AULI_TO_ALR ⟶ Q_NumericValue_PHYSA_LREAL) auf dem Terminal ausgegeben.
4. **Hinweise**: Im XML sind zwei Kommentare enthalten:
   - „hier sind negative Werte möglich!“ – dies bezieht sich auf den Zähler, der bei Rückwärtszählen unter Null gehen kann.
   - „hier gegebenenfalls je einen AX_D_FF einbauen, damit die Events reduziert werden.“ – ein Hinweis für eine mögliche Erweiterung zur Ereignisreduzierung an den Ausgängen.

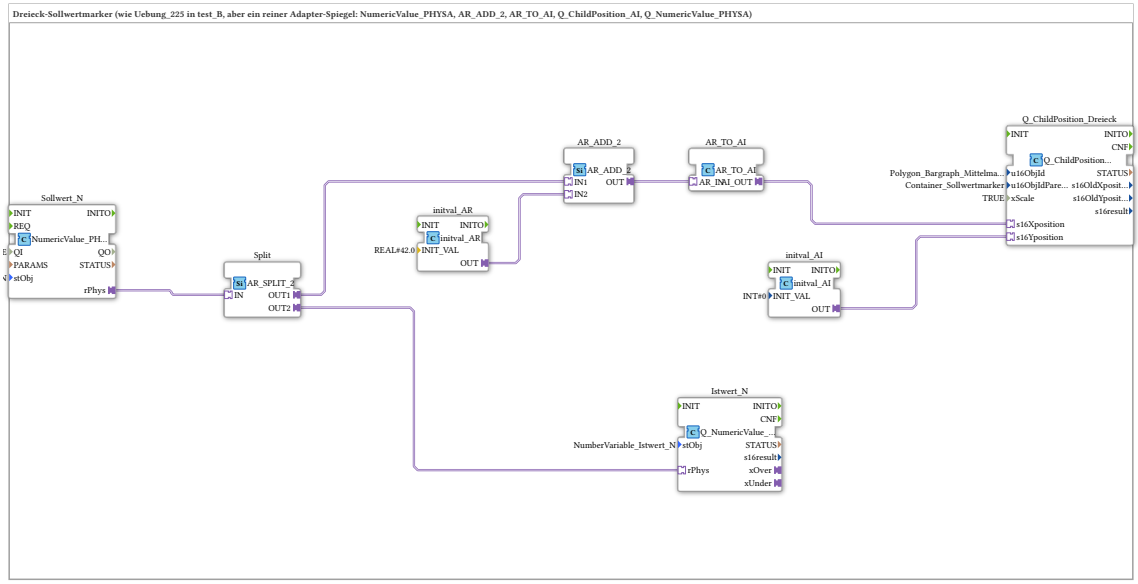
Zusammenfassung

Die Übung 224b ALR demonstriert den Einsatz eines IEC 61131-3 konformen Vorwärts-/Rückwärtszählers (CTUD) in der 4diac-IDE mit Adaptertechnologie. Vier Digitaleingänge steuern den Zähler (Vorwärts, Rückwärts, Reset, Laden), zwei Digitalausgänge geben die Überlauf-/Unterlaufzustände aus, und über einen Terminalbaustein wird der aktuelle Zählerstand als LREAL-Wert visualisiert. Der feste Startwert von 5 wird über einen ULINT-zu-ULI-Konverter bereitgestellt. Die Übung eignet sich für Einsteiger in die IEC 61131-3-Zählerfunktionen und die Adapterkommunikation in 4diac.

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

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

2.1.2.518 Uebung_225_AX: Dreieck-Sollwertmarker als reine Adapter-Kette



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung ist die Adapter-Variante von Übung 225 (test_B): identische Funktion — ein Dreieck-Marker folgt einem Sollwert um einen festen Mittelpunkt-Offset versetzt —, aber **vollständig** über Adapter verdrahtet. Es gibt in der gesamten SubApp kein einziges plain Event- oder DataConnection mehr; sowohl das Sollwert-Lesen und Istwert-Schreiben als auch die komplette Rechenkette zur Dreieck-Bewegung laufen über generische Adapter-Bausteine.

Verwendete Funktionsbausteine (FBs)

- **Sollwert_N** (`isobus::UT::io::NumericValue::NumericValue_PHYSA`): AR-Adapter-Variante von `NumericValue_PHYS`.
  - **Parameter**: `QI = TRUE`, `stObj = NumberVariable_Sollwert_N`.
  - **Erklärung**: Liest den physischen Wert von `InputNumber_Sollwert` und stellt ihn statt über ein plain `rPhys`-Datum als AR-Adapter-Plug (`rPhys`) bereit.
- **Split** (`adapter::events::unidirectional::AR_SPLIT_2`): Adapter-Verteiler für einen REAL-Adapterwert.
  - **Parameter**: keine.
  - **Erklärung**: Ein AR-Adapter-Socket darf nur an genau eine Quelle angeschlossen werden, aber hier braucht der eine gelesene Sollwert zwei Abnehmer (Istwert-Rückschreibung und Center-Addition). `AR_SPLIT_2` verteilt den einen AR-Eingang sauber auf `OUT1`/`OUT2`.
- **AR_ADD_2** (`adapter::iec61131::arithmetic::AR_ADD_2`): Adapter-Version der IEC-61131-Addition.
  - **Parameter**: keine.
  - **Erklärung**: Addiert zwei REAL-Adapterwerte (`IN1`, `IN2`) und liefert die Summe als AR-Adapter-Ausgang `OUT`. Hier: Sollwert + Center-Offset.
- **initval_AR** (`adapter::types::unidirectional::AR::initval::initval_AR`): Konstantengeber für einen AR-Adapterwert.
  - **Parameter**: `INIT_VAL = REAL#42.0`.
  - **Erklärung**: Liefert dauerhaft den festen Center-Offset `42.0` als AR-Adapter-Ausgang — der Adapternative Ersatz für eine plain REAL-Konstante.
- **AR_TO_AI** (`adapter::conversion::unidirectional::AR_TO_AI`): Adapter-Typkonverter REAL → INT.
  - **Parameter**: keine.
  - **Erklärung**: Wandelt den AR-Adapterwert (REAL) in einen AI-Adapterwert (INT) um — die Adapternative Entsprechung von `F_REAL_TO_INT`.
- **initval_AI** (`adapter::types::unidirectional::AI::initval::initval_AI`): Konstantengeber für einen AI-Adapterwert.

  - **Parameter**: `INIT_VAL = 0`.
  - **Erklärung**: Liefert die feste Y-Position `0` als AI-Adapter-Ausgang.
- **Q_ChildPosition_Dreieck** (`isobus::UT::Q::Q_ChildPosition_AI`): Service-Baustein „Change Child Location“ (ISO 11783-6) mit AI-Adapter-Positionseingängen.
  - **Parameter**: `u16ObjId = Polygon_Bargraph_Mittelmarker`, `u16ObjIdParent = Container_Sollwertmarker`, `xScale = TRUE`.
  - **Erklärung**: Bewegt das Dreieck-Objekt innerhalb seines Containers. `s16Xposition` kommt aus der Rechenkette (`AR_TO_AI.AI_OUT`), `s16Yposition` ist über `initval_AI` fest auf `0` verdrahtet — beide als eigener AI-Adapter-Socket statt als plain INT-Eingang.
- **Istwert_N** (`isobus::UT::Q::Q_NumericValue_PHYSA`): AR-Adapter-Variante von `Q_NumericValue_PHYS`.
  - **Parameter**: `stObj = NumberVariable_Istwert_N`.
  - **Erklärung**: Schreibt den unveränderten Sollwert als Istwert zurück, komplett über den AR-Adapter-Socket `rPhys` statt über ein plain `REQ`/`rValue`-Paar.

Sub-Bausteine: keine

Die Übung verwendet ausschließlich generische Adapter-Bausteine direkt auf der obersten Ebene der SubApp — keinen eigenen Composite-FB für die Dreieck-Bewegung, im Gegensatz zu den nachfolgenden Übungen 225b_AX/226_AX/227_AX/228_AX, die dafür `PositionMarkerFSA` bzw. `BargraphSplitFS_AR` einsetzen.

Programmablauf und Verbindungen

1. **Sollwert lesen**: `Sollwert_N` liest `InputNumber_Sollwert` (über `NumberVariable_Sollwert_N`) und liefert ihn als AR-Plug `rPhys`.
2. **Verteilen**: `Sollwert_N.rPhys → Split.IN`. `Split` (`AR_SPLIT_2`) dupliziert den Wert auf zwei unabhängige Ausgänge, da ein Adapter-Socket nicht direkt mehrere Ziele bedienen darf.
3. **Istwert-Zweig**: `Split.OUT2 → Istwert_N.rPhys`. Der Sollwert wird unverändert als Istwert zurückgeschrieben.
4. **Center-Addition**: `Split.OUT1 → AR_ADD_2.IN1`, `initval_AR.OUT (42.0) → AR_ADD_2.IN2`. `AR_ADD_2.OUT` liefert `Sollwert + 42.0`.
5. **Typkonvertierung**: `AR_ADD_2.OUT → AR_TO_AI.AR_IN`. `AR_TO_AI` wandelt das REAL-Ergebnis in ein INT-Adapter-Signal um.
6. **Dreieck bewegen**: `AR_TO_AI.AI_OUT → Q_ChildPosition_Dreieck.s16Xposition` setzt die X-Position; `initval_AI.OUT (0) → Q_ChildPosition_Dreieck.s16Yposition` hält die Y-Position konstant. Das Dreieck (`Polygon_Bargraph_Mittelmarker`) verschiebt sich dadurch horizontal im Container `Container_Sollwertmarker`, um den Sollwert plus 42 Pixel Mittenversatz.
7. Alle Verbindungen laufen ausschließlich über `<AdapterConnections>` — es existiert keine einzige plain Event- oder DataConnection.

Zusammenfassung

Übung 225_AX zeigt, dass sich eine komplette Signalverarbeitungskette — Lesen, Verteilen, Rechnen, Typkonvertieren, Schreiben — vollständig mit generischen Adapter-Bausteinen abbilden lässt, ohne einen einzigen eigenen Composite-Baustein zu schreiben. Funktional ist sie identisch zur klassischen Lösung in Übung 225 (test_B), unterscheidet sich aber grundlegend im Verdrahtungsstil: Statt loser Event-/DataConnections zwischen einzelnen FBs entsteht eine durchgängige Adapter-Pipeline aus `AR_SPLIT_2`, `AR_ADD_2`, `initval_AR`, `AR_TO_AI` und `initval_AI`. Diese Übung bildet die Grundlage für 225b_AX, das dieselbe Aufgabe mit einem wiederverwendbaren Composite-Baustein (`PositionMarkerFSA`) statt der losen Adapter-Kette löst.

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

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

2.1.2.519 Übung 225 AX: Dreieck-Sollwertmarker mit AR-Adapter (reine Adapter-Kette)

Thema: Virtual-Terminal-Objektpositionierung — reine AR/AI-Adapterkette

Adapter-Variante von Übung 225 (test_B): identische Funktion, aber **vollständig** über Adapter verdrahtet — kein einziges plain Event/DataConnection in der SubApp. Sollwert-Lesen und Istwert-Schreiben laufen über `NumericValue_PHYSA`/`Q_NumericValue_PHYSA`, die Dreieck-Bewegung über eine reine Kette generischer Adapter-Bausteine statt eines eigenen Composite-FBs.

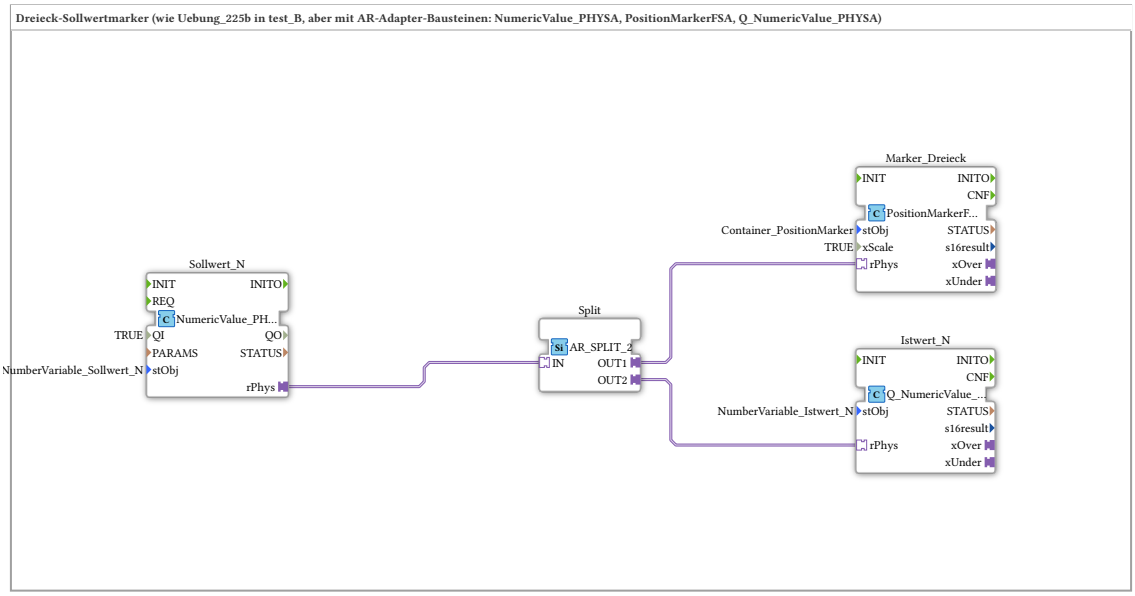
Verdrahtung

- `Sollwert_N` (`NumericValue_PHYSA`) liest `InputNumber_Sollwert`, liefert den Wert als AR-Plug `rPhys`.
- `Split` (`AR_SPLIT_2`) verteilt diesen einen AR-Wert sauber auf zwei Verbraucher (ein Adapter darf nicht direkt auf mehrere Ziele zeigen — siehe `iec61499-creator`-Skill, Regel 8):
  - `Split.OUT2 → Istwert_N.rPhys` — Istwert unverändert zurückschreiben (`Q_NumericValue_PHYSA`).
  - `Split.OUT1 → AR_ADD_2.IN1` — Sollwert in die Center-Addition.
- `AR_ADD_2` addiert den Center-Offset (`initval_AR`, konstant `REAL#42.0`) rein im Adapter-Bereich.
- `AR_TO_AI` konvertiert das Ergebnis von einem AR- (REAL) in ein AI- (INT) Adapter-Signal — die Adapter-native Entsprechung von `F_REAL_TO_INT`.
- `Q_ChildPosition_Dreieck` (`Q_ChildPosition_AI`) bewegt das Dreieck: `s16Xposition` kommt von `AR_TO_AI.AI_OUT`, `s16Yposition` ist über `initval_AI` (konstant `0`) fest verdrahtet — beide als eigener AI-Adapter-Socket statt als plain INT-Eingang.

Referenzlösung

`Uebung_225_AX.SUB` — validiert gegen `subapptype.xsd`. Verwendete Bausteine: `isobus::UT::io::NumericValue::NumericValue_PHYSA`, `adapter::events::unidirectional::AR_SPLIT_2`, `adapter::iec61131::arithmetic::AR_ADD_2`, `adapter::types::unidirectional::AR::initval::initval_AR`, `adapter::conversion::unidirectional::AR_TO_AI`, `isobus::UT::Q::Q_ChildPosition_AI`, `adapter::types::unidirectional::AI::initval::initval_AI`, `isobus::UT::Q::Q_NumericValue_PHYSA`.

2.1.2.520 Uebung_225b_AX: Dreieck-Sollwertmarker mit PositionMarkerFSA



• — ○ • — ○ • — ○ •

Einleitung

Diese Übung ist die Adapter-Variante von Übung 225b (test_B): identische Funktion — ein geklammerter Dreieck-Marker folgt einem Sollwert —, aber Sollwert-Lesen und Istwert-Schreiben laufen über die AR-Adapter-Bausteine `NumericValue_PHYSA`/`Q_NumericValue_PHYSA` statt über plain `REQ`/`IND`-Events. Die Dreieck-Bewegung selbst nutzt weiterhin den wiederverwendbaren Composite-Baustein `PositionMarkerFS`, jetzt aber über dessen neuen Adapter-Wrapper `PositionMarkerFSA` angesprochen.

Verwendete Funktionsbausteine (FBs)

- **Sollwert_N** (`isobus::UT::io::NumericValue::NumericValue_PHYSA`): AR-Adapter-Variante von `NumericValue_PHYS`.
  - **Parameter**: `QI = TRUE`, `stObj = NumberVariable_Sollwert_N`.
  - **Erklärung**: Liest den physischen Wert von `InputNumber_Sollwert` und liefert ihn als AR-Adapter-Plug `rPhys`.

- **Split** (`adapter::events::unidirectional::AR_SPLIT_2`): Adapter-Verteiler für einen REAL-Adapterwert.
  - **Parameter**: keine.
  - **Erklärung**: Verteilt den einen gelesenen Sollwert sauber auf die zwei Verbraucher `Marker_Dreieck` und `Istwert_N` — ein Adapter darf nicht direkt auf mehrere Ziele zeigen.
- **Istwert_N** (`isobus::UT::Q::Q_NumericValue_PHYSA`): AR-Adapter-Variante von `Q_NumericValue_PHYS`.
  - **Parameter**: `stObj = NumberVariable_Istwert_N`.
  - **Erklärung**: Schreibt denselben Sollwert unverändert als Istwert zurück, komplett über den AR-Adapter-Socket `rPhys`.

Sub-Bausteine: Marker_Dreieck (`isobus::UT::Q::PositionMarkerFSA`)

`PositionMarkerFS` selbst besitzt keine AR-Adapter-Schnittstelle. Genau wie `Q_NumericValue_PHYSA` den Baustein `Q_NumericValue_PHYS` um einen AR-Socket herum verpackt, kapselt `PositionMarkerFSA` (`isobus::UT::Q`) jetzt `PositionMarkerFS` auf dieselbe Weise:

- **Parameter der Instanz `Marker_Dreieck`**: `stObj = Container_PositionMarker`, `xScale = TRUE`.
- **Interne Verdrahtung** (im `.fbt` selbst, nicht in dieser Übung geändert): Der AR-Adapter-Socket `rPhys` liefert Event (`E1`) und Datum (`D1`) direkt an eine einzige innenliegende Instanz `Inner` vom Typ `PositionMarkerFS` (`rPhys.E1 → Inner.REQ`, `rPhys.D1 → Inner.rValue`) — die Klammer-Logik (min/max-Begrenzung, Center-Offset) wird also **nicht** dupliziert, sondern unverändert wiederverwendet. `Inner.xOver`/`Inner.xUnder` werden als eigene AX-Adapter-Plugs (`xOver`, `xUnder`) nach außen gereicht, exakt wie bei `Q_NumericValue_PHYSA`. `stObj` und `xScale` werden 1:1 an `Inner` durchgereicht.
- **Bausteindatei**: `Ventilsteuerung\4diacIDE-workspace\.lib\isobus-3.0.0\typelib\UT\Q\PositionMarkerFSA.fbt`.

Die bestehenden Bausteine `NumericValue_PHYSA`, `Q_NumericValue_PHYSA` und `PositionMarkerFS` selbst wurden für diesen Wrapper nicht verändert.

Programmablauf und Verbindungen

1. **Sollwert lesen**: `Sollwert_N` liest `InputNumber_Sollwert` (über `NumberVariable_Sollwert_N`) und liefert ihn als AR-Plug `rPhys`.
2. **Verteilen**: `Sollwert_N.rPhys → Split.IN`. `Split` (`AR_SPLIT_2`) dupliziert den Wert auf `OUT1` und `OUT2`.
3. **Dreieck bewegen**: `Split.OUT1 → Marker_Dreieck.rPhys`. Intern klammert `PositionMarkerFSA` den Wert wie gewohnt (min/max, Center-Offset, `xScale`) und positioniert das Dreieck; bei Überschreitung meldet es dies über seine eigenen `xOver`/`xUnder`-Adapter-Plugs (in dieser Übung nicht weiterverdrahtet).
4. **Istwert zurückschreiben**: `Split.OUT2 → Istwert_N.rPhys`. Der unveränderte Sollwert wird als Istwert zurückgeschrieben.
5. Alles läuft ausschließlich über `<AdapterConnections>` — keine einzige plain Event- oder DataConnection, genau wie bei `Uebung_011b1_PHYSA`.

Zusammenfassung

Übung 225b_AX führt die reine Adapter-Kette aus 225_AX auf die Wiederverwendung eines bewährten Composite-Bausteins zurück: Statt Center-Addition, Typkonvertierung und Positionierung als lose Adapter-Bausteine zu verdrahten, übernimmt `PositionMarkerFSA` die komplette Klammer-Logik von `PositionMarkerFS` und reicht sie über einen einzigen AR-Adapter-Socket nach außen. Funktional entspricht die Übung exakt Übung 225b (test_B) — nur der Verdrahtungsstil (Adapter statt plain Events) unterscheidet sich. Das gleiche Wrapper-Muster wird in Übung 226_AX für den Split-Bargraph (`BargraphSplitFS_AR`) wiederverwendet.

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

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

2.1.2.521 Übung 225b AX: Dreieck-Sollwertmarker mit AR-Adapter (PositionMarkerFS)

Thema: Virtual-Terminal-Objektpositionierung — AR-Adapter-Variante

Adapter-Variante von Übung 225b (test_B): identische Funktion, aber Sollwert- Lesen und Istwert-Schreiben laufen über die AR-Adapter-Bausteine (`NumericValue_PHYSA`/ `Q_NumericValue_PHYSA`)

statt über plain `REQ`/`IND`-Events. Die Dreieck-Bewegung nutzt weiterhin den wiederverwendbaren `PositionMarkerFS` — inklusive Klammern und `xOver`/`xUnder` — passend zu 225b.

Warum ein neuer Baustein nötig war

`PositionMarkerFS` hat keine AR-Adapter-Schnittstelle. Genau wie bei `Q_NumericValue_PHYSA`, das `Q_NumericValue_PHYS` um einen AR-Socket herum verpackt, gibt es jetzt `PositionMarkerFSA` (`isobus::UT::Q`): ein neuer, kleiner Wrapper, der **intern eine einzige Instanz von `PositionMarkerFS` aufruft** (keine Duplizierung der Klammer-Logik) und `xOver`/`xUnder` als AX-Adapter-Plugs nach außen reicht — exakt dieselbe Baustein-Form wie `Q_NumericValue_PHYSA`. Die bestehenden Bausteine `NumericValue_PHYSA`/`Q_NumericValue_PHYSA`/`PositionMarkerFS` wurden dafür **nicht** verändert.

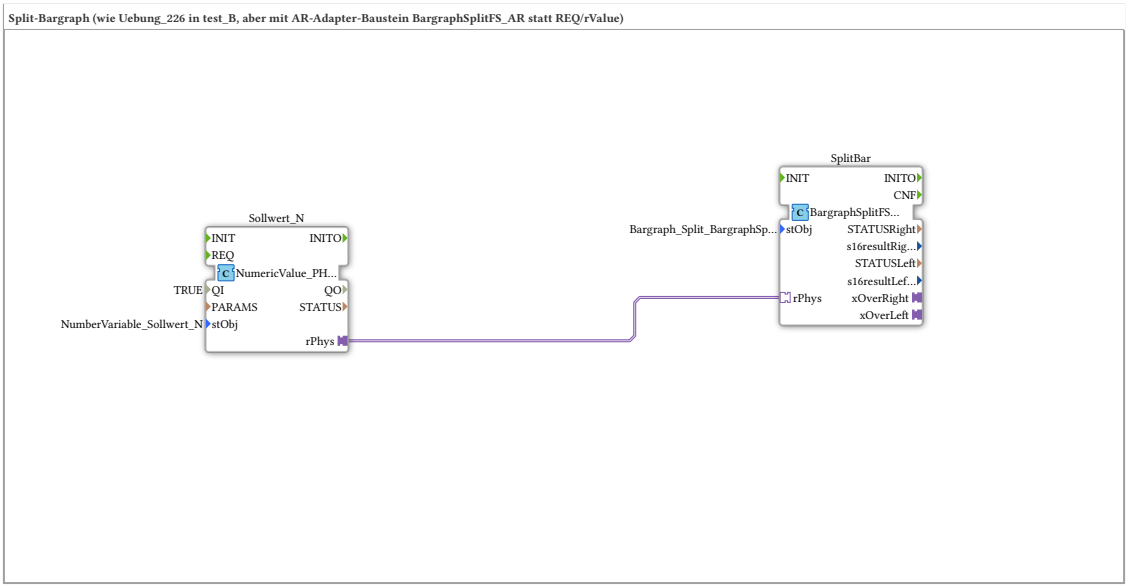
Verdrahtung

- `Sollwert_N` (`NumericValue_PHYSA`) liest `InputNumber_Sollwert`, liefert den Wert als AR-Plug `rPhys`.
- `Split` (`AR_SPLIT_2`) verteilt diesen einen AR-Wert sauber auf zwei Verbraucher (ein Adapter darf nicht direkt auf mehrere Ziele zeigen — siehe `iec61499-creator`-Skill, Regel 8).
- `Marker_Dreieck` (`PositionMarkerFSA`) bewegt das Dreieck, mit Klammerung und `xOver`/`xUnder`.
- `Istwert_N` (`Q_NumericValue_PHYSA`) schreibt denselben Wert als Istwert zurück.
- Alles ausschließlich über `<AdapterConnections>` verdrahtet — keine einzige plain Event-/ DataConnection nötig, genau wie bei `Uebung_011b1_PHYSA`.

Referenzlösung

`Uebung_225b_AX.SUB` — validiert gegen `subapptype.xsd`. Neuer Baustein: `Ventilsteuerung\4diacIDE-workspace\.lib\isobus-3.0.0\typelib\UT\Q\PositionMarkerFSA.fbt`.

2.1.2.522 Uebung_226_AX: Split-Bargraph mit BargraphSplitFS_AR



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung ist die Adapter-Variante von Übung 226 (test_B): identische Funktion — ein Sollwert steuert einen geteilten Bargraphen (links/rechts) an —, aber das Lesen von `InputNumber_Sollwert` läuft über den AR-Adapter-Baustein `NumericValue_PHYSA` statt über ein plain `REQ`/`IND`-Event. Die Bargraph-Ansteuerung selbst nutzt weiterhin den wiederverwendbaren Composite-Baustein `BargraphSplitFS`, jetzt über dessen neuen Adapter-Wrapper `BargraphSplitFS_AR` angesprochen.

Verwendete Funktionsbausteine (FBs)

- **Sollwert_N** (`isobus::UT::io::NumericValue::NumericValue_PHYSA`): AR-Adapter-Variante von `NumericValue_PHYS`.
  - **Parameter**: `QI = TRUE`, `stObj = NumberVariable_Sollwert_N`.

– **Erklärung**: Liest den physischen Wert von `InputNumber_Sollwert` und liefert ihn als AR-Adapter-Plug `rPhys`.

Sub-Bausteine: SplitBar (`isobus::UT::Q::BargraphSplitFS_AR`)

`BargraphSplitFS` selbst besitzt keine AR-Adapter-Schnittstelle. Genau wie `PositionMarkerFSA` (Übung 225b_AX) den Baustein `PositionMarkerFS` um einen AR-Socket herum verpackt, kapselt `BargraphSplitFS_AR` (`isobus::UT::Q`) jetzt `BargraphSplitFS` auf dieselbe Weise:

- **Parameter der Instanz `SplitBar`**: `stObj = Bargraph_Split_BargraphSplit`.
- **Interne Verdrahtung** (im `.fbt` selbst, nicht in dieser Übung geändert): Der AR-Adapter-Socket `rPhys` liefert Event (`E1`) und Datum (`D1`) direkt an eine einzige innenliegende Instanz `Inner` vom Typ `BargraphSplitFS` (`rPhys.E1 → Inner.REQ`, `rPhys.D1 → Inner.rValue`) — die Klammer-Logik für beide Balkenseiten wird also **nicht** dupliziert, sondern unverändert wiederverwendet. `Inner.xOverRight`/`Inner.xOverLeft` werden als eigene AX-Adapter-Plugs (`xOverRight`, `xOverLeft`) nach außen gereicht, exakt wie `xOver`/`xUnder` bei `PositionMarkerFSA`. `stObj` wird 1:1 an `Inner` durchgereicht.
- **Bausteindatei**: `Ventilsteuerung\4diacIDE-workspace\.lib\isobus_signalprocessing_adapter-3.0.0\typelib\UT\Q\`

Die bestehenden Bausteine `NumericValue_PHYSA` und `BargraphSplitFS` selbst wurden für diesen Wrapper nicht verändert.

Programmablauf und Verbindungen

1. **Sollwert lesen**: `Sollwert_N` liest `InputNumber_Sollwert` (über `NumberVariable_Sollwert_N`) und liefert ihn als AR-Plug `rPhys`.
2. **Direkt verbinden**: `Sollwert_N.rPhys → SplitBar.rPhys`. Anders als in Übung 225b_AX (paralleles Istwert-Schreiben) braucht der Sollwert hier nur **einen** Verbraucher — deshalb entfällt `AR_SPLIT_2` vollständig, der Adapter-Socket wird direkt verbunden.
3. **Bargraph ansteuern**: Intern klammert `BargraphSplitFS_AR` den Wert wie gewohnt (gemeinsame Betragsgrenze `r32MaxMagnitude`) und steuert je nach Vorzeichen den linken oder rechten Bargraphen (`Bargraph_Split_BargraphSplit`) an; bei Überschreitung meldet es dies über `xOverRight`/`xOverLeft` (in dieser Übung nicht weiterverdrahtet).
4. Alles läuft ausschließlich über `<AdapterConnections>` — keine einzige plain Event- oder DataConnection, genau wie bei `Uebung_011b1_PHYSA`.

Zusammenfassung

Übung 226_AX zeigt dieselbe Wrapper-Idee wie 225b_AX, diesmal für den Split-Bargraphen: `BargraphSplitFS_AR` verpackt den bewährten `BargraphSplitFS` in einen AR-Adapter-Socket, ohne dessen Klammer-Logik zu duplizieren. Weil hier nur ein einziger Verbraucher den Sollwert benötigt, entfällt der `AR_SPLIT_2`-Verteiler aus 225b_AX — die Verdrahtung wird dadurch sogar einfacher als dort. Funktional entspricht die Übung exakt Übung 226 (test_B); nur der Verdrahtungsstil unterscheidet sich. Übung 227_AX kombiniert anschließend diese Übung mit 225b_AX, sodass ein Sollwert gleichzeitig Dreieck, Bargraph und Istwert bedient.

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

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

2.1.2.523 Übung 226 AX: Split-Bargraph mit AR-Adapter

Thema: Virtual-Terminal-Bargraphen — AR-Adapter-Variante

Adapter-Variante von Übung 226: identische Funktion, aber das Lesen von `InputNumber_Sollwert` läuft über den AR-Adapter-Baustein `NumericValue_PHYSA` statt über ein plain `REQ`/`IND`-Event. Die Bargraph-Ansteuerung nutzt weiterhin den wiederverwendbaren `BargraphSplitFS` — inklusive Klammern und `xOverRight`/`xOverLeft` — passend zu 226.

Warum ein neuer Baustein nötig war

`BargraphSplitFS` hat keine AR-Adapter-Schnittstelle. Genau wie bei `PositionMarkerFSA`, das `PositionMarkerFS` um einen AR-Socket herum verpackt, gibt es jetzt `BargraphSplitFS_AR` (`isobus::UT::Q`): ein neuer, kleiner Wrapper, der **intern eine einzige Instanz von `BargraphSplitFS` aufruft** (keine Duplizierung der Klammer-Logik) und `xOverRight`/`xOverLeft` als AX-Adapter-Plugs nach außen reicht — exakt dieselbe Baustein-Form wie `PositionMarkerFSA`. Die bestehenden Bausteine `NumericValue_PHYSA`/`BargraphSplitFS` wurden dafür **nicht** verändert.

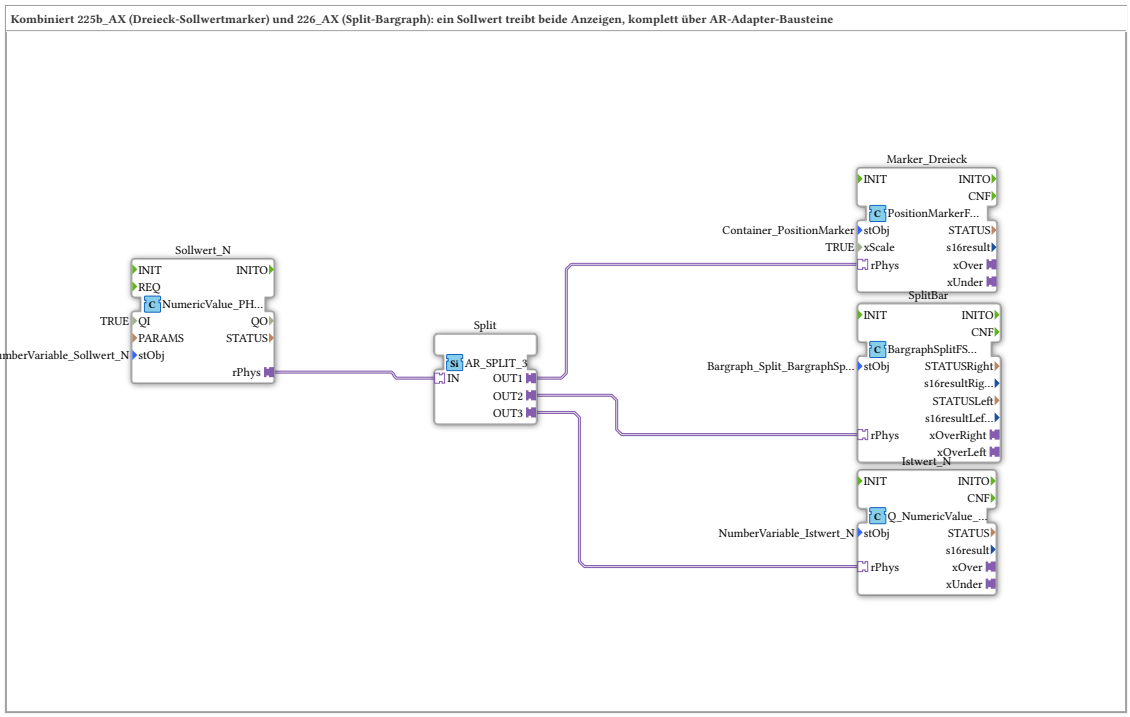
Verdrahtung

- `Sollwert_N` (`NumericValue_PHYSA`) liest `InputNumber_Sollwert`, liefert den Wert als AR-Plug `rPhys`.
- `SplitBar` (`BargraphSplitFS_AR`) steuert beide Bargraphen an, mit Klammerung und `xOverRight`/`xOverLeft`.
- Da hier - anders als bei 225b_AX mit seinem parallelen Istwert-Schreiben - nur **ein** Verbraucher den Sollwert braucht, entfällt `AR_SPLIT_2`: `Sollwert_N.rPhys` verbindet sich direkt mit `SplitBar.rPhys`.
- Alles ausschließlich über `<AdapterConnections>` verdrahtet - keine einzige plain Event-/ DataConnection nötig, genau wie bei `Uebung_011b1_PHYSA`.

Referenzlösung

`Uebung_226_AX.SUB` — validiert gegen `subapptype.xsd`. Neuer Baustein: `Ventilsteuerung\4diacIDE-workspace\.lib\isobus-3.0.0\typelib\UT\Q\BargraphSplitFS_AR.fbt`.

2.1.2.524 Uebung_227_AX: Dreieck-Sollwertmarker und Split-Bargraph aus einer gemeinsamen Quelle



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung kombiniert Übung 225b_AX (Dreieck-Sollwertmarker) und Übung 226_AX (Split-Bargraph): **ein** Sollwert treibt **beide** Anzeigen gleichzeitig sowie die Istwert-Rückschreibung. Anders als die beiden Einzelübungen läuft hier alles konsequent über AR-Adapter-Bausteine — da Übung 226_AX ihren Sollwert bereits nur über den AR-Adapter `NumericValue_PHYSA` liest, müssen für eine gemeinsame Quelle beide Verbraucher denselben Verdrahtungsstil (Adapter) verwenden.

Verwendete Funktionsbausteine (FBs)

- **Sollwert_N** (`isobus::UT::io::NumericValue::NumericValue_PHYSA`): AR-Adapter-Variante von `NumericValue_PHYS`.
  - **Parameter**: `QI = TRUE`, `stObj = NumberVariable_Sollwert_N`.
  - **Erklärung**: Liest `NumberVariable_Sollwert_N` (die an `InputNumber_Sollwert`, VT-Objekt 9000, gebundene Variable) und liefert den Wert als AR-Adapter-Plug `rPhys`.
- **Split** (`adapter::events::unidirectional::AR_SPLIT_3`): Adapter-Verteiler für einen REAL-Adapterwert auf drei Ziele.
  - **Parameter**: keine.

  - **Erklärung**: Verteilt den einen gelesenen Sollwert sauber auf die drei Verbraucher `Marker_Dreieck`, `SplitBar` und `Istwert_N` — ein Adapter darf nicht direkt auf mehrere Ziele zeigen. Gegenüber `AR_SPLIT_2` in 225b_AX kommt hier ein dritter Ausgang hinzu.
- **Istwert_N** (`isobus::UT::Q::Q_NumericValue_PHYSA`): AR-Adapter-Variante von `Q_NumericValue_PHYS`.
  - **Parameter**: `stObj = NumberVariable_Istwert_N`.
  - **Erklärung**: Schreibt denselben Sollwert als Istwert auf `NumberVariable_Istwert` zurück — dies treibt sowohl `InputNumber_Istwert` als auch den bestehenden Einzel-Bargraph-Zeiger.

Sub-Bausteine: Marker_Dreieck (`PositionMarkerFSA`) und SplitBar (`BargraphSplitFS_AR`)

Diese Übung führt keine neuen Bausteintypen ein, sondern kombiniert zwei bereits bekannte Wrapper aus den Vorübungen an derselben Quelle:

- **Marker_Dreieck** (`isobus::UT::Q::PositionMarkerFSA`, `stObj = Container_PositionMarker`, `xScale = TRUE`): bewegt das Dreieck genau wie in Übung 225b_AX beschrieben — intern eine einzige `PositionMarkerFS`-Instanz mit Klammerung und `xOver`/`xUnder`-Plugs.
- **SplitBar** (`isobus::UT::Q::BargraphSplitFS_AR`, `stObj = Bargraph_Split_BargraphSplit`): steuert den Split-Bargraphen genau wie in Übung 226_AX beschrieben — intern eine einzige `BargraphSplitFS`-Instanz mit Klammerung und `xOverRight`/`xOverLeft`-Plugs.

Programmablauf und Verbindungen

1. **Sollwert lesen**: `Sollwert_N` liest `NumberVariable_Sollwert_N` und liefert den Wert als AR-Plug `rPhys`.
2. **Verteilen**: `Sollwert_N.rPhys → Split.IN`. `Split` (`AR_SPLIT_3`) dupliziert den Wert auf drei unabhängige Ausgänge `OUT1`, `OUT2`, `OUT3`.
3. **Dreieck bewegen**: `Split.OUT1 → Marker_Dreieck.rPhys`. Das Dreieck (`Polygon_Bargraph_Mittelmarker` im Container `Container_PositionMarker`) folgt dem Sollwert mit Klammerung und Skalierung (`xScale = TRUE`).
4. **Split-Bargraph ansteuern**: `Split.OUT2 → SplitBar.rPhys`. Je nach Vorzeichen füllt sich der linke oder rechte Bargraph (`Bargraph_Split_links/_rechts`) passend zum Betrag.
5. **Istwert zurückschreiben**: `Split.OUT3 → Istwert_N.rPhys`. Derselbe Sollwert wird unverändert als Istwert auf `NumberVariable_Istwert` zurückgeschrieben; das treibt gleichzeitig `InputNumber_Istwert` und den bereits bestehenden Einzel-Bargraph-Zeiger.
6. Alles läuft ausschließlich über `<AdapterConnections>` — keine einzige plain Event- oder DataConnection.

Am echten Terminal zeigt sich das Zusammenspiel: Wird der Sollwert geändert, folgt das Dreieck, der Split-Bargraph füllt die passende Seite, und sowohl das Istwert-Feld als auch der bestehende Einzel-Bargraph-Zeiger zeigen denselben Wert an.

Zusammenfassung

Übung 227_AX zeigt, wie sich mehrere unabhängige VT-Anzeigen ohne neue Bausteintypen aus einer einzigen Sollwertquelle speisen lassen, sobald alle Verbraucher konsequent über AR-Adapter angesprochen werden: `AR_SPLIT_3` verteilt den einen gelesenen Wert auf `PositionMarkerFSA`, `BargraphSplitFS_AR` und `Q_NumericValue_PHYSA`, ohne Duplizierung der Lese-Logik. Die Übung baut direkt auf 225b_AX und 226_AX auf und bildet die Grundlage für Übung 228_AX, die zusätzlich eine vierte Verwendung des Sollwerts — eine Farblogik für das Dreieck — einführt.

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

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

2.1.2.525 Übung 227: Kombinierte Übung — Dreieck-Sollwertmarker + Split-Bargraph (AR-Adapter)

Thema: mehrere VT-Anzeigen aus einer Sollwertquelle — durchgängig über AR-Adapter

Kombiniert Übung 225b (Dreieck-Sollwertmarker) und Übung 226 (Split-Bargraph): **ein** Sollwert treibt **beide** Anzeigen gleichzeitig. Anders als die beiden Einzelübungen läuft hier alles konsequent über AR-Adapter-Bausteine (wie `225b_AX`), da `226_AX` seinen Sollwert bereits nur über den AR-Adapter

(NumericValue_PHYSA) liest — für eine gemeinsame Quelle müssen beide Verbraucher denselben Verdrahtungsstil (Adapter) verwenden.

Situationsbeschreibung

Derselbe Sollwert (InputNumber_Sollwert, VT-Objekt 9000, Bereich −42…+42) steuert gleichzeitig:

- das Dreieck (Polygon_Bargraph_Mittelmarker) im Container Container_Sollwertmarker,
- den Split-Bargraph (Bargraph_Split_links/_rechts),
- und schreibt wie bisher den Istwert (NumberVariable_Istwert) zurück.

Funktionsbeschreibung

- **Sollwert lesen:** Sollwert_N (NumericValue_PHYSA) liest NumberVariable_Sollwert_N (VT-Objekt 21000 — die an InputNumber_Sollwert gebundene Variable, siehe Übung 225 Beschreibung zur Begründung) und liefert den Wert als AR-Plug rPhys.
- **Verteilen:** Split (AR_SPLIT_3) verteilt diesen einen AR-Wert sauber auf drei Verbraucher (ein Adapter darf nicht direkt auf mehrere Ziele zeigen — siehe iec61499-creator-Skill, Regel 8).
- **Dreieck bewegen:** Marker_Dreieck (PositionMarkerFSA, xScale := TRUE) — mit Klammerung und xOver/xUnder als AX-Plugs.
- **Split-Bargraph ansteuern:** SplitBar (BargraphSplitFS_AR) — mit Klammerung und xOverRight/xOverLeft als AX-Plugs.
- **Istwert zurückschreiben:** Istwert_N (Q_NumericValue_PHYSA) schreibt denselben Wert auf NumberVariable_Istwert zurück (treibt InputNumber_Istwert und den bestehenden Einzel-Bargraph-Zeiger).
- Alles ausschließlich über <AdapterConnections> verdrahtet — keine einzige plain Event-/ DataConnection nötig.

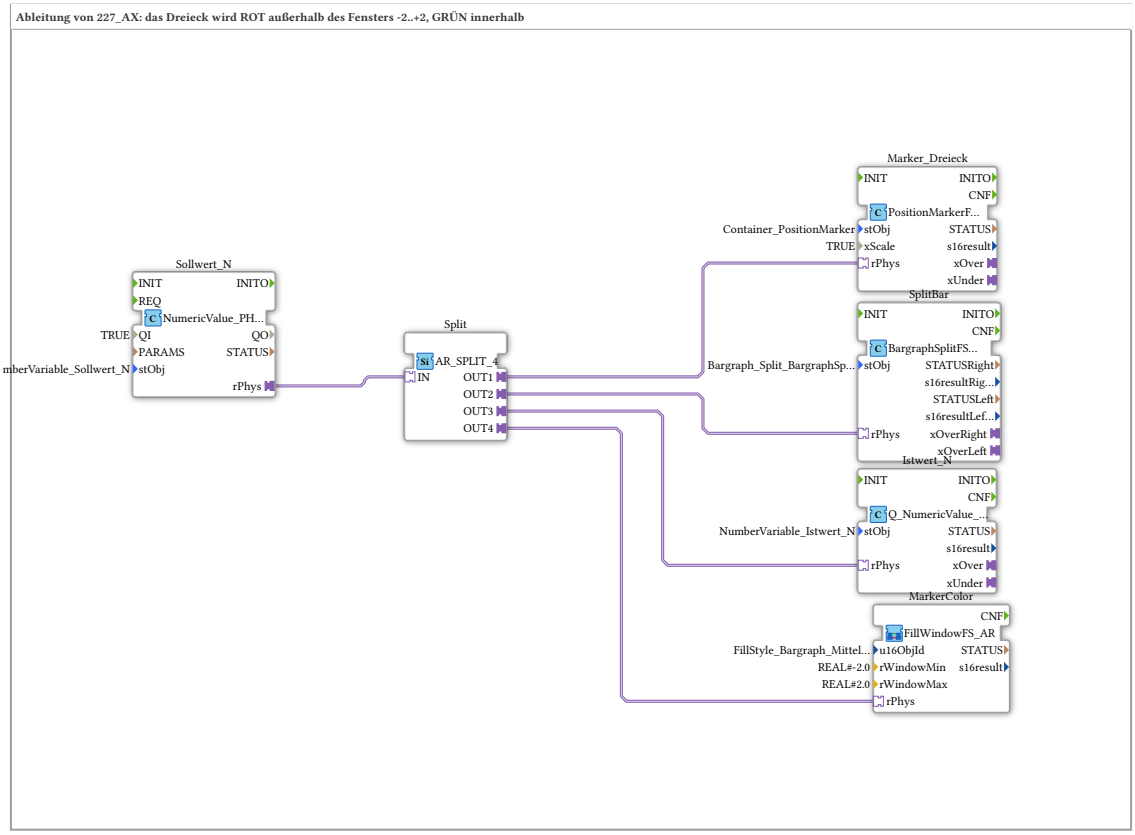
Arbeitsauftrag

1. Legen Sie die SubApp Uebung_227_AX an (bereits als Referenzlösung vorhanden).
2. Lesen Sie NumberVariable_Sollwert_N mit NumericValue_PHYSA.
3. Verteilen Sie den Wert mit AR_SPLIT_3 auf drei Ziele.
4. Bewegen Sie das Dreieck mit PositionMarkerFSA (stObj := Container_PositionMarker, xScale := TRUE).
5. Steuern Sie den Split-Bargraph mit BargraphSplitFS_AR (stObj := Bargraph_Split_BargraphSplit).
6. Schreiben Sie den Istwert mit Q_NumericValue_PHYSA (stObj := NumberVariable_Istwert_N) zurück.
7. Am echten Terminal testen: Sollwert ändern ⟶ Dreieck folgt, Split-Bargraph füllt die passende Seite, Istwert-Feld und der bestehende Einzel-Bargraph-Zeiger zeigen denselben Wert.

Referenzlösung

Uebung_227_AX.SUB — validiert gegen subapptype.xsd. Verwendet ausschließlich bereits bestehende Bausteine (PositionMarkerFSA, BargraphSplitFS_AR, AR_SPLIT_3) — keine neuen Typen für diese Übung nötig.

2.1.2.526 Uebung_228_AX: Fenster-Farbrückmeldung am Dreieck (Change Fill Attributes, ISO 11783-6 F.32)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung ist eine Ableitung von `Uebung_227_AX.md`: der Sollwert bewegt weiterhin ein Dreieck (Positionsmarker) und einen Split-Bargraph und wird als Istwert zurückgeschrieben, alles über AR-Adapter. Zusätzlich wird das Dreieck innerhalb des Fensters −2…+2 grün und außerhalb rot gefärbt. Die Übung demonstriert damit den Einsatz von Change Fill Attributes (ISO 11783-6, F.32) als einfache Statusanzeige auf dem VT-Terminal.

Verwendete Funktionsbausteine (FBs)

- **Sollwert_N**: Terminal-Eingabe für den Sollwert
  - **Typ**: `isobus::UT::io::NumericValue::NumericValue_PHYSA`
  - **Parameter**: `QI = TRUE`, `stObj = NumberVariable_Sollwert_N`
  - **Erklärung**: Liest den vom Bediener eingegebenen Sollwert vom VT-Terminal und stellt ihn als physikalischen Wert (`rPhys`, AR-Adapter) bereit.
- **Split**: Verteiler für den Sollwert
  - **Typ**: `adapter::events::unidirectional::AR_SPLIT_4`
  - **Parameter**: keine
  - **Erklärung**: Verteilt den einen eingehenden AR-Sollwert auf vier Verbraucher (eine Stelle mehr als in `Uebung_227_AX.md`): Dreieck-Position, Split-Bargraph, Istwert-Rückschreibung und die Farblogik.
- **Marker_Dreieck**: Positionsmarker (Dreieck) auf dem Terminal
  - **Typ**: `isobus::UT::Q::PositionMarkerFSA`
  - **Parameter**: `stObj = Container_PositionMarker`, `xScale = TRUE`
  - **Erklärung**: Positioniert das Dreieck-Objekt entsprechend dem Sollwert.
- **SplitBar**: Split-Bargraph auf dem Terminal
  - **Typ**: `isobus::UT::Q::BargraphSplitFS_AR`
  - **Parameter**: `stObj = Bargraph_Split_BargraphSplit`
  - **Erklärung**: Stellt den Sollwert zusätzlich als geteilten Bargraph dar.
- **Istwert_N**: Terminal-Ausgabe des Istwerts

  - **Typ**: `isobus::UT::Q::Q_NumericValue_PHYSA`
  - **Parameter**: `stObj = NumberVariable_Istwert_N`
  - **Erklärung**: Schreibt den (unverändert durchgereichten) Sollwert als Istwert auf das Terminal zurück.
- **MarkerColor**: Fenster-Farblogik für das Dreieck
  - **Typ**: `MyLib::sys::FillWindowFS_AR` (SubApp-Instanz)
  - **Parameter**: `u16ObjId = FillStyle_Bargraph_Mittelmarker_Gruen`, `rWindowMin = REAL#-2.0`, `rWindowMax = REAL#2.0`
  - **Erklärung**: Erhält den Sollwert über einen AR-Socket und färbt das FillAttributes-Objekt `FillStyle_Bargraph_Mittelmarker_Gruen` per Change Fill Attributes (ISO 11783-6 F.32) grün, solange der Sollwert im Fenster −2…+2 liegt, sonst rot.

Sub-Bausteine: `FillWindowFS_AR` (SubAppType)

`FillWindowFS_AR` (`MyLib_AX-1.0.0\typelib\sys\FillWindowFS_AR.SUB`, Version 2.0) färbt ein FillAttributes-Objekt grün innerhalb von `[rWindowMin, rWindowMax]` und rot außerhalb. Er besteht ausschließlich aus vorhandenen Bausteinen, ohne eigene ST-Logik:

- **WindowMinConst/WindowMaxConst** (`initval_AR`): stellen die Fenstergrenzen `rWindowMin`/`rWindowMax` als AR-Werte bereit.
- **CENTER_CMP** (`AR_DEADBAND_COMPARE`): erhält den Sollwert (`rPhys` an `Y`) und die Grenzen (`WINDOW_MIN`, `WINDOW_MAX`) und liefert `CENTERED` für „im Fenster“.
- **ColorSel** (`AX_SEL`, `IN0 = COLOR_RED`, `IN1 = COLOR_GREEN`): wählt anhand von `CENTERED` (an `G`) zwischen Rot und Grün.
- **Inner** (`Q_FillAttributes`, `u8FillType = USINT#2`, `u16FillPatternId = ID_NULL`): sendet die gewählte Farbe (`u8FillColour`) als Change-Fill-Attributes-Kommando an das per `u16ObjId` referenzierte VT-Objekt.

Programmablauf und Verbindungen

1. `Sollwert_N.rPhys` ⟶ `Split.IN` (AdapterConnection): Der Sollwert speist den Verteiler `AR_SPLIT_4`.
2. `Split.OUT1` ⟶ `Marker_Dreieck.rPhys`: bewegt das Dreieck entsprechend dem Sollwert.
3. `Split.OUT2` ⟶ `SplitBar.rPhys`: stellt denselben Sollwert zusätzlich als Split-Bargraph dar.
4. `Split.OUT3` ⟶ `Istwert_N.rPhys`: schreibt den Sollwert unverändert als Istwert auf das Terminal zurück.
5. `Split.OUT4` ⟶ `MarkerColor.rPhys`: speist denselben Sollwert in die Fenster-Farblogik.
6. Innerhalb von `MarkerColor` (`FillWindowFS_AR`) vergleicht `AR_DEADBAND_COMPARE` den Sollwert mit dem Fenster `-2.0…+2.0`; `ColorSel` wählt `COLOR_GREEN` bzw. `COLOR_RED`, und `ColorSel.CNF` löst `Q_FillAttributes.REQ` aus.
7. `Q_FillAttributes` sendet ein Change-Fill-Attributes-Kommando (ISO 11783-6 F.32) an `FillStyle_Bargraph_Mittelmarker_Gruen`, wodurch sich die Füllfarbe des Dreiecks live ändert: grün innerhalb, rot außerhalb des Fensters −2…+2.

Zusammenfassung

Die Übung 228 erweitert den aus `Uebung_227_AX.md` bekannten Aufbau (Dreieck, Split-Bargraph, Istwert, alles über AR-Adapter) um eine vierte Verbrauchsstelle des Sollwerts: eine Fenster-basierte Farbrückmeldung, die das Dreieck grün oder rot einfärbt. Sie zeigt, wie Change Fill Attributes (ISO 11783-6 F.32) als Statusanzeige genutzt werden kann und wie sich die Fensterlogik als SubApp aus vorhandenen Adapter-Bausteinen zusammensetzen lässt, ohne eigene ST-Logik.

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

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

2.1.2.527 Übung 228: Kombinierte Übung mit Fenster-Farbrückmeldung

Thema: Change Fill Attributes (ISO 11783-6 F.32) als Statusanzeige

Ableitung von Übung 227: identischer Aufbau (Dreieck + Split-Bargraph + Istwert, alles über AR-Adapter), zusätzlich wird das Dreieck **rot**, sobald der Sollwert das Fenster **−2…+2** verlässt, und **grün**, solange er darin liegt.

Situationsbeschreibung

Der Sollwert bewegt weiterhin das Dreieck und den Split-Bargraph und wird als Istwert zurückgeschrieben (siehe Übung 227). Zusätzlich färbt sich das Dreieck selbst je nach Sollwert: grün in der Nähe von 0 (Fenster −2…+2), rot außerhalb.

Funktionsbeschreibung

- **Warum ein neuer Baustein nötig war:** Das Dreieck (`Polygon_Bargraph_Mittelmarker`) bezieht seine Füllfarbe von einem eigenen, nur für dieses Objekt genutzten `FillStyle`-Objekt (`FillStyle_Bargraph_Mittelmarker_Gruen`, VT-Objekt 25000, `FillType=2` = feste Farbe, kein Muster). Die Farbe eines FillAttributes-Objekts wird per **Change Fill Attributes** (ISO 11783-6 F.32, `Q_FillAttributes`) gesetzt. Da dieses FillAttributes-Objekt exklusiv vom Dreieck verwendet wird, ändert das Umfärben keine anderen Objekte.
- Der neue Baustein `FillWindowFS_AR` (`isobus::UT::Q`) kapselt das — als `SubAppType`, nach dem Vorbild von `MyLib_AX-1.0.0\typelib\sys\GreenRedBackground1_AX.SUB`, statt als eigener `FBType` mit einer ST-Hilfsfunktion: er nimmt den Sollwert über einen AR-Socket entgegen, verteilt ihn mit `AR_SPLIT_2` auf `AR_GE` (`>= rWindowMin`) und `AR_LE` (`<= rWindowMax`), UND-verknüpft beide Ergebnisse mit `AX_AND_2` und wählt darüber mit `AX_SEL` (`G` = „im Fenster?“) zwischen `COLOR_RED`/`COLOR_GREEN` — ausschließlich vorhandene generische Adapter-Bausteine, keine eigene ST-Logik. `FillType`/`FillPatternId` bleiben dabei fest auf „feste Farbe, kein Muster“, genau wie im Pool konfiguriert. Ein sauberer, generischer `Q_FillAttributes`-Adapter-Wrapper (als `FBType`) ist als spätere Verfeinerung sinnvoll, aber für den Moment bewusst zurückgestellt.
- **Verteilen:** `Split` (`AR_SPLIT_4`, eine Stelle mehr als in Übung 227) verteilt den einen AR-Sollwert jetzt auf vier Verbraucher: Dreieck-Position, Split-Bargraph, Istwert-Rückschreibung und die neue Farblogik.

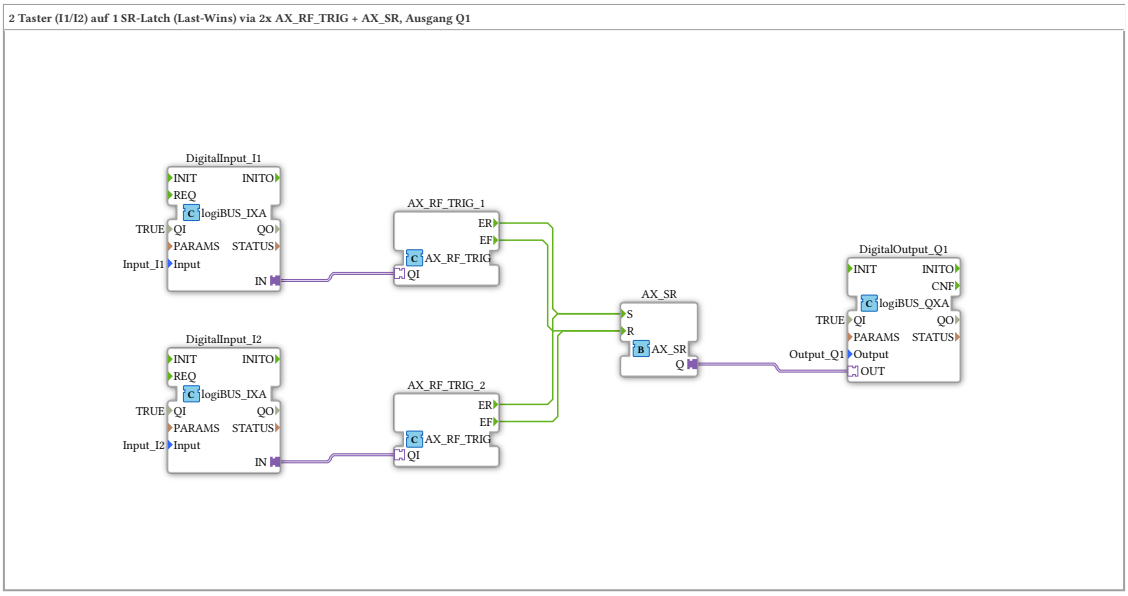
Arbeitsauftrag

1. Legen Sie die SubApp `Uebung_228_AX` an (bereits als Referenzlösung vorhanden).
2. Verteilen Sie den gelesenen Sollwert mit `AR_SPLIT_4` auf vier Ziele (wie Übung 227, plus eins).
3. Färben Sie das Dreieck mit `MarkerColor` (`FillWindowFS_AR`, `u16ObjId := FillStyle_Bargraph_Mittelmarker_Gruen`, `rWindowMin := -2.0`, `rWindowMax := 2.0`).
4. Am echten Terminal testen: Sollwert zwischen −2 und +2 ⟶ Dreieck grün; außerhalb ⟶ Dreieck rot.

Referenzlösung

`Uebung_228_AX.SUB` — validiert gegen `subapptype.xsd`. Neuer Baustein: `Ventilsteuerung\4diacIDE-workspace\.lib\isobus-3.0.0\typelib\UT\Q\FillWindowFS_AR.SUB`.

2.1.2.528 Uebung_229_AX: Zwei Taster, ein SR-Latch (Last-Wins) mit AX_RF_TRIG



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung schaltet zwei tastende Taster (`I1`, `I2`) auf denselben digitalen Ausgang `Q1`. Da beide Taster tastend sind (kein Dauersignal), wird eine Set-Reset-Verriegelung (`AX_SR`) benötigt. Die Übung baut dabei bewusst eine „Last-Wins"-Verriegelung: nicht nur das Drücken, sondern auch das Loslassen beider Taster wirkt auf denselben Latch — welche Flanke zuletzt auftrat, bestimmt den Zustand von `Q1`.

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_I1**, **DigitalInput_I2**: logiBUS Digitaleingänge
  - **Typ**: `logiBUS::io::DI::logiBUS_IXA`
  - **Parameter**: `QI = TRUE`, `Input = Input_I1` bzw. `Input_I2`
  - **Erklärung**: Koppeln die beiden physischen Taster-Eingänge an die Adapterschnittstelle.
- **AX_RF_TRIG_1**, **AX_RF_TRIG_2**: Flankenerkennung je Taster
  - **Typ**: `adapter::events::unidirectional::AX_RF_TRIG`
  - **Parameter**: keine
  - **Erklärung**: Jeder physische Eingang wird von einer eigenen Instanz beobachtet. Jede Instanz meldet die steigende Flanke (`ER`, Taster gedrückt) und die fallende Flanke (`EF`, Taster losgelassen) als zwei separate Events.
- **AX_SR**: gemeinsame Set-Reset-Verriegelung
  - **Typ**: `adapter::events::unidirectional::AX_SR`
  - **Parameter**: keine
  - **Erklärung**: Set-dominante Verriegelung. Beide `AX_RF_TRIG`-Instanzen speisen denselben `AX_SR`: beide steigenden Flanken (`ER` von I1 und I2) laufen auf `AX_SR.S`, beide fallenden Flanken (`EF` von I1 und I2) auf `AX_SR.R`. Das ist in 4diac legal, da EventConnections — anders als DataConnections — mehrere Quellen auf ein Ziel erlauben; es entsteht eine reine ODER-Verknüpfung ohne zusätzlichen Baustein.
- **DigitalOutput_Q1**: logiBUS Digitalausgang
  - **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
  - **Parameter**: `QI = TRUE`, `Output = Output_Q1`
  - **Erklärung**: Gibt den Zustand von `AX_SR.Q` an den physischen Ausgang `Q1` weiter.

Sub-Bausteine: keine

Die Übung verwendet keine weiteren Unterbausteine, alle FBs sind direkt auf der obersten Ebene der SubApp angeordnet.

Programmablauf und Verbindungen

1. `DigitalInput_I1.IN` ⟶ `AX_RF_TRIG_1.QI` und `DigitalInput_I2.IN` ⟶ `AX_RF_TRIG_2.QI` (AdapterConnections): die physischen Tasterzustände werden an die jeweilige Flankenerkennung übergeben.
2. `AX_RF_TRIG_1.ER` ⟶ `AX_SR.S` und `AX_RF_TRIG_2.ER` ⟶ `AX_SR.S` (EventConnections): die steigenden Flanken beider Taster setzen den Latch.
3. `AX_RF_TRIG_1.EF` ⟶ `AX_SR.R` und `AX_RF_TRIG_2.EF` ⟶ `AX_SR.R` (EventConnections): die fallenden Flanken beider Taster setzen den Latch zurück.
4. `AX_SR.Q` ⟶ `DigitalOutput_Q1.OUT` (AdapterConnection): der Latch-Zustand wird auf den physischen Ausgang `Q1` gelegt.

Wichtig — Last-Wins, nicht „hält nach Loslassen": Der Ausgang `Q1` folgt nicht dem Zustand eines bestimmten Tasters, sondern immer der zuletzt aufgetretenen Flanke, egal an welchem Taster. Drückt man `I1`, geht `Q1` EIN. Lässt man `I1` wieder los, geht `Q1` sofort wieder AUS — auch wenn `I2` zu diesem Zeitpunkt noch gedrückt gehalten wird. Ebenso setzt `I2`s steigende Flanke `Q1` auf EIN, und `I2`s fallende Flanke schaltet `Q1` sofort wieder AUS, unabhängig vom Zustand von `I1`. Jedes Loslassen eines beliebigen der beiden Taster schaltet `Q1` also sofort AUS, selbst wenn der andere Taster noch gehalten wird — es handelt sich ausdrücklich nicht um ein Verhalten, bei dem `Q1` EIN bleibt, solange irgendein Taster gehalten wird.

Zusammenfassung

Die Übung 229 demonstriert, wie zwei unabhängige Ereignisquellen über einfache EventConnections auf denselben Set- bzw. Reset-Eingang eines Latches geführt werden können, ohne einen zusätzlichen ODER-Baustein zu benötigen. Das Ergebnis ist eine bewusst als „Last-Wins" konzipierte Verriegelung, deren Zustand stets der zuletzt aufgetretenen Flanke folgt — ein Verhalten, das von einer klassischen Halte-Logik zu unterscheiden ist. Siehe `Uebung_230_AX.md` für dieselbe Funktion, umgesetzt mit ASR-Adapter-Bausteinen anstelle loser Event-Verkabelung.

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

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

2.1.2.529 Übung 229: Zwei Taster, ein SR-Latch (Last-Wins) — klassisch mit AX_RF_TRIG

Thema: Flankenerkennung + Set/Reset-Verriegelung, zwei Quellen auf einer Verriegelung

Situationsbeschreibung

Zwei tastende Taster `I1` und `I2` sollen denselben Ausgang `Q1` schalten. Da beide Taster tastend sind (kein Dauersignal), braucht es eine Verriegelung (SR-Latch). Diese Übung baut bewusst eine „Last-Wins"-Verriegelung: nicht nur das Drücken (steigende Flanke), sondern auch das Loslassen (fallende Flanke) BEIDER Taster wirkt auf denselben Latch - welche Flanke (Drücken oder Loslassen, egal an welchem Taster) zuletzt auftrat, bestimmt den Zustand von `Q1`.

Funktionsbeschreibung

- **Flankenerkennung:** Jeder physische Eingang wird über einen eigenen `AX_RF_TRIG` beobachtet - dieser meldet eine steigende Flanke (`ER`, Taster gedrückt) und eine fallende Flanke (`EF`, Taster losgelassen) als separate Events.
- **Zusammenführen zweier Quellen auf einer Verriegelung:** Beide `AX_RF_TRIG`-Instanzen speisen denselben `AX_SR` (Set-Reset-Latch): `I1`s steigende Flanke UND `I2`s steigende Flanke laufen beide auf `AX_SR.S`, beide fallenden Flanken auf `AX_SR.R`. Das ist legal in 4diac, da EventConnections (anders als DataConnections) mehrere Quellen auf ein Ziel erlauben - eine reine ODER-Verknüpfung ohne zusätzlichen Baustein.
- **Ausgang:** `AX_SR.Q` treibt direkt `Output_Q1`.

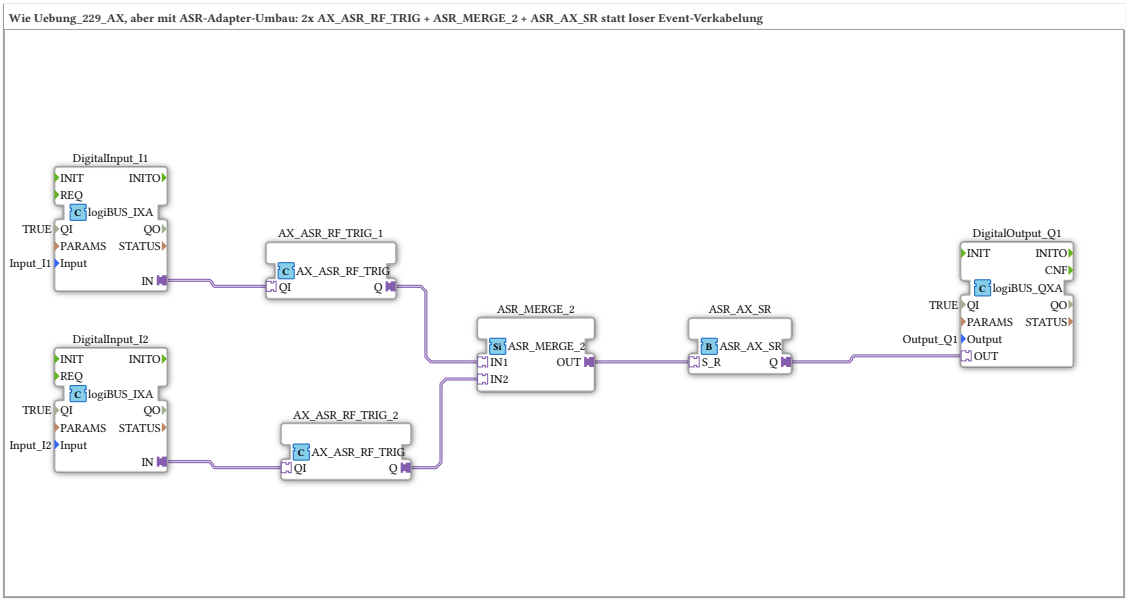
Arbeitsauftrag

1. Legen Sie die SubApp `Uebung_229_AX` an (bereits als Referenzlösung vorhanden).
2. Verbinden Sie `Input_I1`/`Input_I2` über je einen `AX_RF_TRIG` mit dem gemeinsamen `AX_SR`.
3. Testen Sie: `I1` drücken → `Q1` EIN; `I1` loslassen → `Q1` wieder AUS (die fallende Flanke von `I1` wirkt ebenfalls auf `AX_SR.R`). Ebenso setzt `I2`s steigende Flanke `Q1` auf EIN, `I2`s fallende Flanke wieder auf AUS. Der Ausgang folgt also nicht dem Tasterzustand, sondern immer der zuletzt aufgetretenen Flanke (Drücken oder Loslassen, an welchem Taster auch immer).

Referenzlösung

`Uebung_229_AX.SUB`. Siehe auch `Uebung_230_AX` für dieselbe Funktion, umgebaut auf die Adapter-Bausteine `AX_ASR_RF_TRIG`/`ASR_MERGE_2`/`ASR_AX_SR` statt loser Event-Verkabelung.

2.1.2.530 Uebung_230_AX: Zwei Taster, ein SR-Latch (Last-Wins) mit ASR-Adaptern



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung realisiert exakt dieselbe Funktion wie `Uebung_229_AX.md` (zwei Taster `I1`/`I2` schalten gemeinsam `Q1` über ein Last-Wins-SR-Latch), setzt sie jedoch komplett mit den neueren `ASR`-Adapter-Bausteinen um: statt jede Flanke als losen Event zu verkabeln, trägt ein `ASR`-Adapter Set und Reset gebündelt in einem Stecker. Die Übung demonstriert damit, wie sich dieselbe Logik typsicher über AdapterConnections statt über rohe EventConnections abbilden lässt.

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_I1**, **DigitalInput_I2**: logiBUS Digitaleingänge
  - **Typ**: `logiBUS::io::DI::logiBUS_IXA`
  - **Parameter**: `QI = TRUE`, `Input = Input_I1` bzw. `Input_I2`
  - **Erklärung**: Koppeln die beiden physischen Taster-Eingänge an die Adapterschnittstelle.
- **AX_ASR_RF_TRIG_1**, **AX_ASR_RF_TRIG_2**: Flankenerkennung je Taster mit ASR-Ausgang
  - **Typ**: `adapter::events::unidirectional::AX_ASR_RF_TRIG`
  - **Parameter**: keine
  - **Erklärung**: Ersetzt `AX_RF_TRIG` aus `Uebung_229_AX.md`. Intern derselbe `E_RF_TRIG`-Kern (steigende/fallende Flanke), aber die beiden Events werden nicht als lose EventOutputs (`ER`/`EF`) exportiert, sondern direkt in einen gebündelten `ASR`-Adapter-Ausgang geschrieben (`Q.SET`/`Q.RESET`).
- **ASR_MERGE_2**: Zusammenführung der beiden ASR-Quellen
  - **Typ**: `adapter::events::unidirectional::ASR_MERGE_2`
  - **Parameter**: keine
  - **Erklärung**: Ersetzt die vier losen EventConnections aus `Uebung_229_AX.md`. Nimmt die beiden `ASR`-Adapter der beiden Taster entgegen und führt SET mit SET, RESET mit RESET zusammen — dieselbe ODER-Verknüpfung wie zuvor, jetzt aber als eigener, wiederverwendbarer, typsicherer Baustein statt Ad-hoc-Verkabelung.
- **ASR_AX_SR**: gemeinsame Set-Reset-Verriegelung mit gebündeltem ASR-Eingang
  - **Typ**: `adapter::events::unidirectional::ASR_AX_SR`
  - **Parameter**: keine
  - **Erklärung**: Ersetzt `AX_SR` aus `Uebung_229_AX.md`. Identischer ECC (Set-dominant, START→SET→RESET→SET), aber Set und Reset kommen gebündelt über einen einzigen `ASR`-Socket (`S_R`) statt als zwei lose EventInputs.
- **DigitalOutput_Q1**: logiBUS Digitalausgang
  - **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
  - **Parameter**: `QI = TRUE`, `Output = Output_Q1`
  - **Erklärung**: Gibt den Zustand von `ASR_AX_SR.Q` an den physischen Ausgang `Q1` weiter.

Sub-Bausteine: keine

Die Übung verwendet keine weiteren Unterbausteine, alle FBs sind direkt auf der obersten Ebene der SubApp angeordnet.

Programmablauf und Verbindungen

Diese Version verwendet ausschließlich AdapterConnections — es existiert kein einziges EventConnections-Element mehr:

1. `DigitalInput_I1.IN` → `AX_ASR_RF_TRIG_1.QI` und `DigitalInput_I2.IN` → `AX_ASR_RF_TRIG_2.QI`: die physischen Tasterzustände werden an die jeweilige Flankenerkennung übergeben.
2. `AX_ASR_RF_TRIG_1.Q` → `ASR_MERGE_2.IN1` und `AX_ASR_RF_TRIG_2.Q` → `ASR_MERGE_2.IN2`: die gebündelten Set/Reset-Signale beider Taster laufen in den Merge-Baustein.
3. `ASR_MERGE_2.OUT` → `ASR_AX_SR.S_R`: das zusammengeführte ASR-Signal steuert die gemeinsame Verriegelung.
4. `ASR_AX_SR.Q` → `DigitalOutput_Q1.OUT`: der Latch-Zustand wird auf den physischen Ausgang `Q1` gelegt.

Verhalten identisch zu `Uebung_229_AX.md`: Es handelt sich weiterhin um eine „Last-Wins"-Verriegelung, keine Halte-Logik. Drückt man `I1`, geht `Q1` EIN; lässt man `I1` los, geht `Q1` sofort wieder AUS, selbst wenn `I2` noch gedrückt gehalten wird — und umgekehrt. `ASR_MERGE_2` bildet dieselbe ODER-Verknüpfung von SET mit SET und RESET mit RESET ab, die in `Uebung_229_AX.md` durch die parallel geschalteten EventConnections entstand. Das Verhalten ist von außen ununterscheidbar von `Uebung_229_AX.md`; es unterscheidet sich ausschließlich der Bauplan dahinter — ein Vergleich der beiden SubApps im 4diac-

Editor zeigt, dass `Uebung_229_AX.md` ein `EventConnections`-Element mit vier Verbindungen besitzt, während `Uebung_230_AX.md` gar keines hat.

Zusammenfassung

Die Übung 230 zeigt, wie sich eine mit losen Event-Verbindungen aufgebaute Logik (`Uebung_229_AX.md`) vollständig auf typsichere Adapter-Bausteine umbauen lässt, ohne das Verhalten zu verändern. `AX_ASR_RF_TRIG`, `ASR_MERGE_2` und `ASR_AX_SR` kapseln dieselbe Flankenerkennung, ODER-Verknüpfung und Set-Reset-Logik wie zuvor, jetzt aber als wiederverwendbare, über AdapterConnections verdrahtete Bausteine. Die Übung verdeutlicht damit den Mehrwert von Adaptertechnologie gegenüber Ad-hoc-Event-Verkabelung bei gleichbleibender Funktion.

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

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

2.1.2.531 Übung 230: Zwei Taster, ein SR-Latch (Last-Wins) — Umbau mit ASR-Adapter

Thema: Dieselbe Funktion wie Übung 229, aber mit gebündelten Set/Reset-Adaptern statt loser Events

Situationsbeschreibung

Exakt dieselbe Funktion wie `Uebung_229_AX` (zwei Taster `I1`/`I2` schalten gemeinsam `Q1` über ein Last-Wins-SR-Latch), aber mit den neueren Adapter-Bausteinen umgesetzt: statt jede Flanke als losen Event zu verkabeln, trägt ein `ASR`-Adapter Set UND Reset gebündelt in einem Stecker.

Funktionsbeschreibung

- **`AX_ASR_RF_TRIG`** ersetzt `AX_RF_TRIG`: intern derselbe `E_RF_TRIG`-Kern (steigende/fallende Flanke), aber die beiden Events werden direkt in einen `ASR`-Adapter-Ausgang geschrieben (`Q.SET`/`Q.RESET`) statt als zwei lose EventOutputs (`ER`/`EF`) exportiert zu werden.
- **`ASR_MERGE_2`** ersetzt die 4 losen EventConnections aus Übung 229: er nimmt die beiden `ASR`-Adapter der beiden Taster entgegen und führt SET mit SET, RESET mit RESET zusammen - dieselbe ODER-Verknüpfung wie zuvor, jetzt aber als eigener, wiederverwendbarer, typsicherer Baustein statt Ad-hoc-Verkabelung.
- **`ASR_AX_SR`** ersetzt `AX_SR`: identischer ECC (Set-dominant, START→SET→RESET→SET), aber Set und Reset kommen gebündelt über einen `ASR`-Socket (`S_R`) statt als zwei lose EventInputs.
- Ergebnis: exakt dasselbe Verhalten wie Übung 229, aber komplett ohne rohe EventConnections - alle Verbindungen laufen über AdapterConnections.

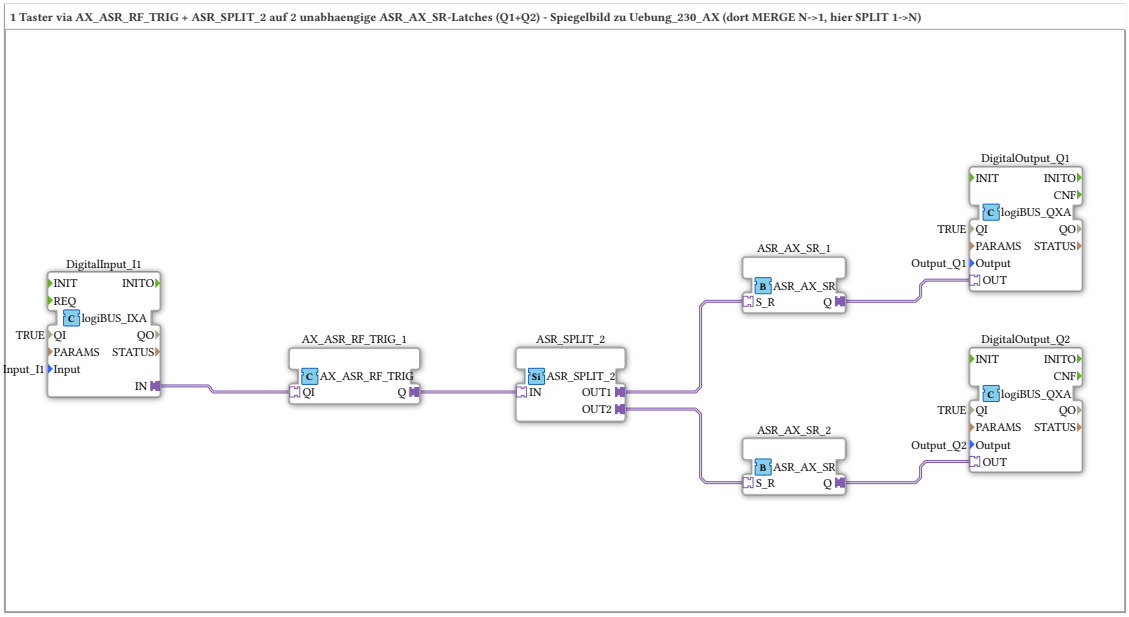
Arbeitsauftrag

1. Legen Sie die SubApp `Uebung_230_AX` an (bereits als Referenzlösung vorhanden).
2. Verbinden Sie `Input_I1`/`Input_I2` über je einen `AX_ASR_RF_TRIG` mit `ASR_MERGE_2`, dessen Ausgang mit `ASR_AX_SR.S_R`, und `ASR_AX_SR.Q` mit `Output_Q1`.
3. Testen Sie identisch zu Übung 229 - das Verhalten muss ununterscheidbar sein, nur die Bausteine dahinter sind andere.
4. Vergleichen Sie die beiden SubApps im 4diac-Editor: Übung 229 hat ein `EventConnections`-Element mit 4 Verbindungen, Übung 230 hat gar keins - alles ist über Adapter typisiert.

Referenzlösung

`Uebung_230_AX.SUB`. Die verwendeten Bausteine liegen in `Ventilsteuerung\4diacIDE-workspace\.lib\adapter-3.0.0\typelib\events\unidirectional\BOOL\AX_ASR_RF_TRIG.fbt`, `...\BOOL\ASR_AX_SR.fbt` und `...\EVENT\ASR_MERGE_2.fbt`.

2.1.2.532 Uebung_231_AX: Ein Taster über ASR_SPLIT_2 auf zwei unabhängige Latches (Fan-out)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung ist das Spiegelbild von `Uebung_230_AX.md`: dort führte `ASR_MERGE_2` zwei Quellen zu einem gemeinsamen Latch zusammen (N⟶1, Fan-in). Hier steuert genau ein Taster (`I1`) über `ASR_SPLIT_2` zwei unabhängige Latches gleichzeitig (1⟶N, Fan-out). Die Übung demonstriert, dass ein Adapter-Plug strikt punkt-zu-punkt ist und deshalb — genau spiegelbildlich zum Merge-Fall — ein Split-Baustein nötig ist, um einen einzelnen ASR-Stream verlustfrei auf mehrere Ziele zu verteilen.

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_I1**: logiBUS Digitaleingang
  - **Typ**: `logiBUS::io::DI::logiBUS_IXA`
  - **Parameter**: `QI = TRUE`, `Input = Input_I1`
  - **Erklärung**: Koppelt den physischen Taster-Eingang an die Adapterschnittstelle.
- **AX_ASR_RF_TRIG_1**: Flankenerkennung mit ASR-Ausgang
  - **Typ**: `adapter::events::unidirectional::AX_ASR_RF_TRIG`
  - **Parameter**: keine
  - **Erklärung**: Taster drücken (steigende Flanke) ⟶ `SET`, loslassen (fallende Flanke) ⟶ `RESET`, als ein einziges gebündeltes `ASR`-Signal — derselbe Baustein wie in `Uebung_230_AX.md`.
- **ASR_SPLIT_2**: Verteiler für den einen ASR-Stream
  - **Typ**: `adapter::events::unidirectional::ASR_SPLIT_2`
  - **Parameter**: keine
  - **Erklärung**: Ein Adapter-Plug (`AX_ASR_RF_TRIG_1.Q`) ist strikt punkt-zu-punkt — er kann nicht direkt gleichzeitig an zwei `ASR_AX_SR.S_R`-Sockets angeschlossen werden. Genau dieselbe Grundregel, die in `Uebung_230_AX.md` `ASR_MERGE_2` nötig machte (mehrere Quellen können nicht direkt auf einen Adapter-Socket zeigen — anders als bei den rohen `EventConnections` in `Uebung_229_AX.md`, wo das erlaubt ist), erzwingt hier umgekehrt `ASR_SPLIT_2` (eine Quelle kann nicht direkt zwei Sockets gleichzeitig bedienen). `ASR_SPLIT_2` dupliziert den einen ASR-Stream verlustfrei auf zwei Ausgänge.
- **ASR_AX_SR_1**, **ASR_AX_SR_2**: zwei unabhängige Set-Reset-Verriegelungen
  - **Typ**: `adapter::events::unidirectional::ASR_AX_SR`
  - **Parameter**: keine
  - **Erklärung**: Zwei getrennte Instanzen desselben Latch-Bausteins wie in `Uebung_230_AX.md`. Beide erhalten über `ASR_SPLIT_2` exakt dieselben SET/RESET-Kommandos zur exakt gleichen Zeit und laufen daher synchron, obwohl sie zwei komplett unabhängige Bausteine mit je eigenem Ausgang sind — es gibt keine direkte Verbindung zwischen ihnen.
- **DigitalOutput_Q1**, **DigitalOutput_Q2**: logiBUS Digitalausgänge

- **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
- **Parameter**: `QI = TRUE`, `Output = Output_Q1` bzw. `Output_Q2`
- **Erklärung**: Geben die Zustände von `ASR_AX_SR_1.Q` bzw. `ASR_AX_SR_2.Q` an die physischen Ausgänge weiter.

Sub-Bausteine: keine

Die Übung verwendet keine weiteren Unterbausteine, alle FBs sind direkt auf der obersten Ebene der SubApp angeordnet.

Programmablauf und Verbindungen

1. `DigitalInput_I1.IN` ⟶ `AX_ASR_RF_TRIG_1.QI`: der physische Tasterzustand wird an die Flankenerkennung übergeben.
2. `AX_ASR_RF_TRIG_1.Q` ⟶ `ASR_SPLIT_2.IN`: das gebündelte SET/RESET-Signal läuft in den Verteiler.
3. `ASR_SPLIT_2.OUT1` ⟶ `ASR_AX_SR_1.S_R` und `ASR_SPLIT_2.OUT2` ⟶ `ASR_AX_SR_2.S_R`: derselbe ASR-Stream wird verlustfrei auf beide Latches dupliziert.
4. `ASR_AX_SR_1.Q` ⟶ `DigitalOutput_Q1.OUT` und `ASR_AX_SR_2.Q` ⟶ `DigitalOutput_Q2.OUT`: beide Latch-Zustände werden auf die jeweiligen physischen Ausgänge gelegt.

Verhalten: Wird `I1` gedrückt und gehalten, gehen `Q1` UND `Q2` gleichzeitig EIN. Lässt man `I1` los, gehen beide gleichzeitig AUS. `ASR_AX_SR_1` und `ASR_AX_SR_2` sind zwei getrennte Instanzen, die ausschließlich über den gemeinsamen `ASR_SPLIT_2`-Ursprung gekoppelt sind — es besteht keine direkte Verbindung zwischen ihnen.

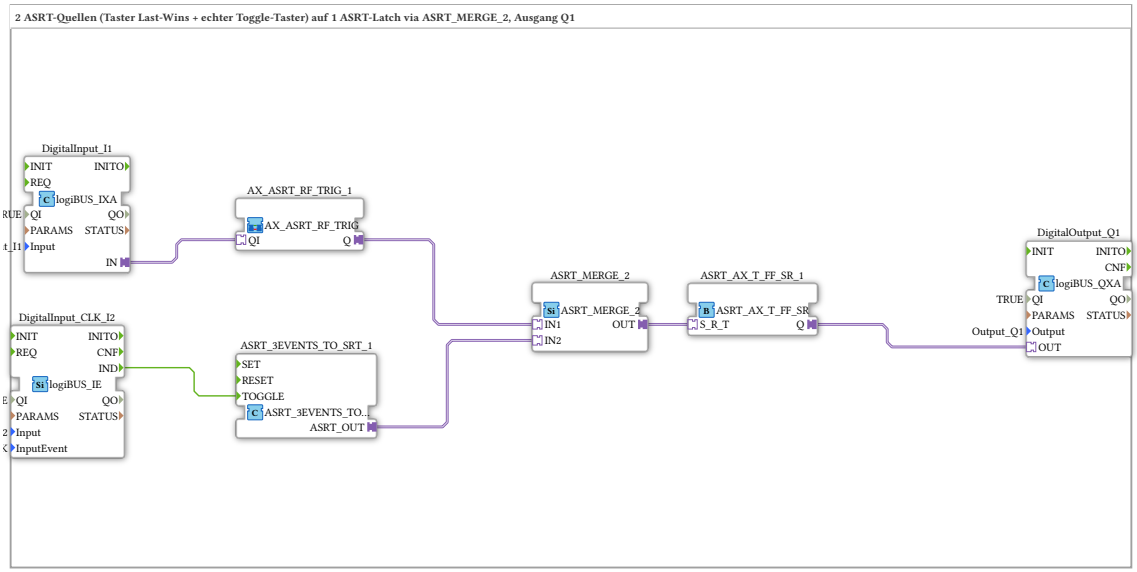
Zusammenfassung

Die Übung 231 zeigt die zu `Uebung_230_AX.md` spiegelbildliche Situation: während dort mehrere Quellen über `ASR_MERGE_2` auf einen gemeinsamen Latch zusammengeführt wurden (Fan-in), verteilt hier `ASR_SPLIT_2` eine einzelne Quelle auf mehrere unabhängige Latches (Fan-out). Beide Übungen zusammen verdeutlichen die grundlegende Punkt-zu-Punkt-Natur von Adapter-Plugs und -Sockets in 4diac und die dafür vorgesehenen Split-/Merge-Bausteine, mit denen sich diese Einschränkung sauber auflösen lässt.

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

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

2.1.2.533 Uebung_232_AX: Zwei ASRT-Quellen (Taster + echter Toggle-Taster) auf ein ASRT-Latch



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung ist ein direktes Geschwister zu `Uebung_229_AX.md`/`Uebung_230_AX.md`, verwendet jedoch durchgängig `ASRT` (Set/Reset/Toggle) statt `ASR` (nur Set/Reset): zwei unabhängige Quellen liefern je einen Teil der SET/RESET/TOGGLE-Semantik, `ASRT_MERGE_2` führt beide zu einem gemeinsamen `ASRT_AX_T_FF_SR`-Latch zusammen. Die Übung zeigt damit, wie sich ein klassischer Last-Wins-Taster und ein echter Toggle-Taster unabhängig voneinander auf denselben Ausgang wirken lassen.

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_I1**: logiBUS Digitaleingang für den klassischen Taster
  - **Typ**: `logiBUS::io::DI::logiBUS_IXA`
  - **Parameter**: `QI = TRUE`, `Input = Input_I1`
  - **Erklärung**: Koppelt den physischen Taster-Eingang `I1` an die Adapterschnittstelle.
- **DigitalInput_CLK_I2**: logiBUS Eingang mit Klick-Erkennung für den Toggle-Taster
  - **Typ**: `logiBUS::io::DI::logiBUS_IE`
  - **Parameter**: `QI = TRUE`, `Input = Input_I2`, `InputEvent = BUTTON_SINGLE_CLICK`
  - **Erklärung**: Erkennt einen einzelnen Klick auf `I2` und meldet ihn als Event `IND`.
- **AX_ASRT_RF_TRIG_1**: Quelle 1 — klassischer Taster mit Last-Wins-Verhalten
  - **Typ**: `MyLib::sys::AX_ASRT_RF_TRIG` (SubApp-Instanz)
  - **Parameter**: keine
  - **Erklärung**: Drücken (steigende Flanke) → `SET`, Loslassen (fallende Flanke) → `RESET`, ausgegeben als gebündeltes `ASRT`-Signal. `TOGGLE` feuert hier nie — nicht grundsätzlich, sondern weil `AX_ASRT_RF_TRIG`s `TOGGLE_IN` intern unverdrahtet bleibt (siehe „Sub-Bausteine“ unten).
- **ASRT_3EVENTS_TO_SRT_1**: Quelle 2 — echter Toggle-Taster
  - **Typ**: `adapter::conversion::unidirectional::ASRT_3EVENTS_TO_SRT`
  - **Parameter**: keine
  - **Erklärung**: Nur der `TOGGLE`-Eingang ist verdrahtet (gespeist von `I2`s `BUTTON_SINGLE_CLICK`), `SET`/`RESET` bleiben unbenutzt. Erzeugt daraus ein gebündeltes `ASRT`-Signal, das ausschließlich die TOGGLE-Schiene trägt.
- **ASRT_MERGE_2**: Zusammenführung der beiden ASRT-Quellen
  - **Typ**: `adapter::events::unidirectional::ASRT_MERGE_2`
  - **Parameter**: keine
  - **Erklärung**: Vereinigt beide ASRT-Streams zu einem gemeinsamen Signal — dieselbe ODER-Zusammenführung wie bei `ASR_MERGE_2` in `Uebung_230_AX.md`, jetzt inklusive der dritten (Toggle-)Schiene.
- **ASRT_AX_T_FF_SR_1**: gemeinsame Set-Reset-Toggle-Verriegelung
  - **Typ**: `adapter::events::unidirectional::ASRT_AX_T_FF_SR`
  - **Parameter**: keine
  - **Erklärung**: Nimmt das zusammengeführte SET/RESET/TOGGLE-Signal über den `S_R_T`-Socket entgegen und steuert daraus den Ausgangszustand `Q`.
- **DigitalOutput_Q1**: logiBUS Digitalausgang
  - **Typ**: `logiBUS::io::DQ::logiBUS_QXA`
  - **Parameter**: `QI = TRUE`, `Output = Output_Q1`
  - **Erklärung**: Gibt den Zustand von `ASRT_AX_T_FF_SR_1.Q` an den physischen Ausgang `Q1` weiter.

Sub-Bausteine: `AX_ASRT_RF_TRIG` (Composite-SubApp)

`AX_ASRT_RF_TRIG` liegt als Composite-SubApp in `MyLib_AX-1.0.0/typelib/sys/AX_ASRT_RF_TRIG.SUB` und wurde aus bereits vorhandenen Bausteinen zusammengesetzt statt als neuer Low-Level-FBType: dem bestehenden `AX_ASR_RF_TRIG` (Flankenerkennung mit ASR-Ausgang, siehe `Uebung_230_AX.md`) plus `ASRT_SR_AE_TO_SRT`, dessen `SR_IN` direkt aus dem ASR-Ausgang gespeist wird und dessen `TOGGLE_IN` unverdrahtet bleibt. So entsteht ein ASRT-Signal, das nur SET/RESET trägt und niemals TOGGLE feuert.

Programmablauf und Verbindungen

1. `DigitalInput_I1.IN` → `AX_ASRT_RF_TRIG_1.QI` (AdapterConnection): der physische Tasterzustand von `I1` speist Quelle 1.
2. `DigitalInput_CLK_I2.IND` → `ASRT_3EVENTS_TO_SRT_1.TOGGLE` (EventConnection): jeder Einzelklick auf `I2` löst ein TOGGLE-Ereignis in Quelle 2 aus.
3. `AX_ASRT_RF_TRIG_1.Q` → `ASRT_MERGE_2.IN1` und `ASRT_3EVENTS_TO_SRT_1.ASRT_OUT` → `ASRT_MERGE_2.IN2` (AdapterConnections): beide ASRT-Quellen laufen in den Merge-Baustein.
4. `ASRT_MERGE_2.OUT` → `ASRT_AX_T_FF_SR_1.S_R_T`: das zusammengeführte SET/RESET/TOGGLE-Signal steuert die gemeinsame Verriegelung.

5. `ASRT_AX_T_FF_SR_1.Q` ⟶ `DigitalOutput_Q1.OUT`: der Latch-Zustand wird auf den physischen Ausgang `Q1` gelegt.

Verhalten: `I1` liefert weiterhin dasselbe Last-Wins-Verhalten wie in `Uebung_229_AX.md`/`Uebung_230_AX.md`: Drücken löst `SET` aus (`Q1` ⟶ EIN), Loslassen löst `RESET` aus (`Q1` ⟶ AUS). Der Toggle-Taster `I2` wirkt davon unabhängig: Jeder Klick auf `I2` togglet den aktuellen Zustand von `Q1` sofort um, unabhängig vom physischen Zustand von `I1`. Das bedeutet konkret: Hält man `I1` gedrückt (`Q1` also gerade durch `SET` auf EIN), kann ein Klick auf `I2` `Q1` trotzdem auf AUS umschalten, obwohl `I1` weiterhin gedrückt bleibt — `Q1` folgt hier also nicht permanent dem Haltezustand von `I1`, sondern jeweils dem zuletzt eingetroffenen SET-, RESET- oder TOGGLE-Ereignis, unabhängig davon, von welcher der beiden Quellen es stammt.

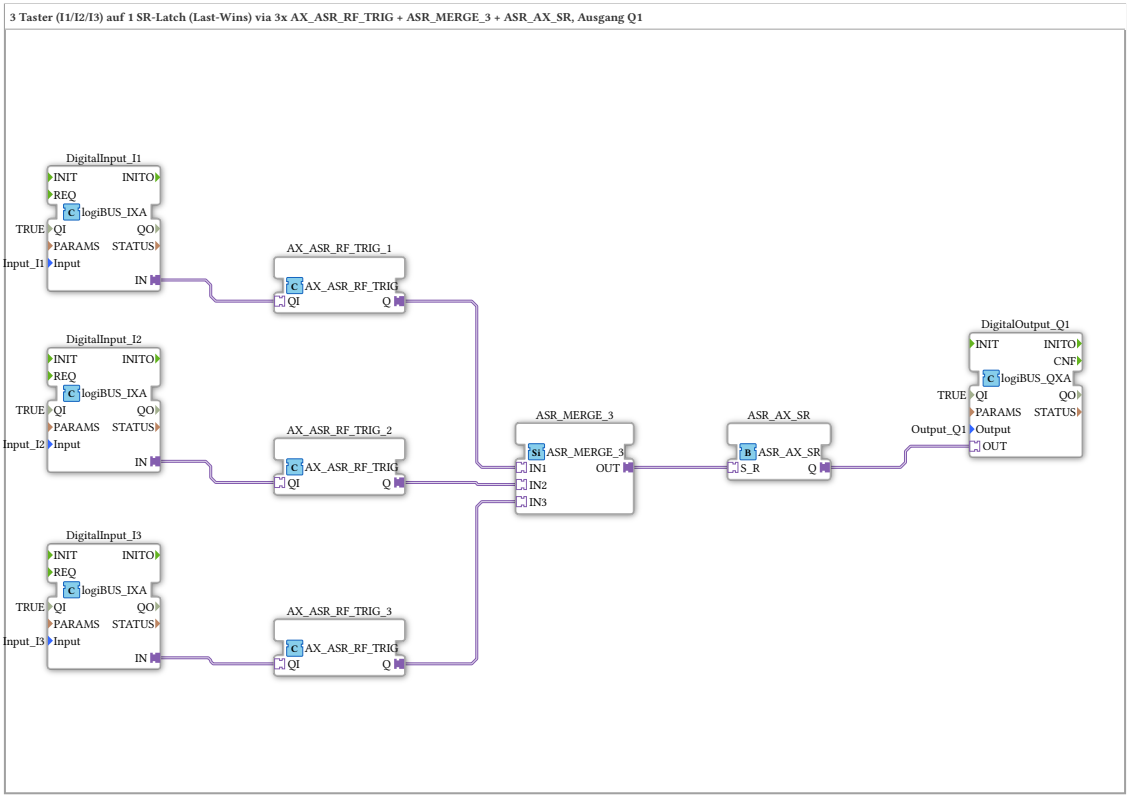
Zusammenfassung

Die Übung 232 kombiniert zwei grundverschiedene Bedienphilosophien auf einem gemeinsamen Latch: einen klassischen Last-Wins-Taster (SET/RESET aus Flanken, wie in `Uebung_229_AX.md`/`Uebung_230_AX.md`) und einen echten Toggle-Taster (ausschließlich TOGGLE, ausgelöst durch einen Einzelklick), zusammengeführt über `ASRT_MERGE_2` auf `ASRT_AX_T_FF_SR`. Sie zeigt, dass sich die ASR-Adaptertechnik aus den Vorgänger-Übungen um eine dritte, unabhängige Toggle-Schiene erweitern lässt, ohne die bestehende Merge-Logik grundlegend zu ändern, und dass verschiedene Bedienkonzepte so sauber auf einen gemeinsamen Ausgang gebracht werden können.

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

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

2.1.2.534 Uebung_233_AX: 3 Taster über ASR_MERGE_3 auf 1 gemeinsames SR-Latch (Last-Wins)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung ist die direkte Fortsetzung von `Uebung_230_AX` („2 Taster, Last-Wins“ via `ASR_MERGE_2`): hier werden **drei** unabhängige Taster über `ASR_MERGE_3` zu einem gemeinsamen Latch zusammengeführt. Sie

zeigt, dass die MERGE-Familie (ASR_MERGE_2..7) kein reiner 2-Input-Spezialfall ist, sondern für mehrere Quellen bis zu sieben Eingänge skaliert.

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_I1**, **DigitalInput_I2**, **DigitalInput_I3**: `logiBUS::io::DI::logiBUS_IXA`
  - **Parameter**: `QI = TRUE`, `Input = Input_I1` / `Input_I2` / `Input_I3`
  - **Erklärung**: Koppeln die drei physischen Taster an die Adapterschnittstellen der nachfolgenden AX_ASR_RF_TRIG-Bausteine.
- **AX_ASR_RF_TRIG_1**, **AX_ASR_RF_TRIG_2**, **AX_ASR_RF_TRIG_3**: `adapter::events::unidirectional::AX_ASR_RF_TRIG`
  - **Parameter**: keine
  - **Erklärung**: Beobachten jeweils einen Taster und wandeln steigende Flanke (Drücken) und fallende Flanke (Loslassen) in ein gebündeltes ASR-Signal (SET/RESET) um.
- **ASR_MERGE_3**: `adapter::events::unidirectional::ASR_MERGE_3`
  - **Parameter**: keine
  - **Erklärung**: Führt alle drei ASR-Streams zu einem gemeinsamen zusammen — strukturell identisch zu ASR_MERGE_2 aus Uebung_230_AX, nur mit einem dritten Socket (IN3). Die Verknüpfung bleibt eine reine ODER-Verknüpfung über alle Quellen.
- **ASR_AX_SR**: `adapter::events::unidirectional::ASR_AX_SR`
  - **Parameter**: keine
  - **Erklärung**: Latcht das gemergte Signal wie in 229/230/233 gehabt: Last-Wins — welcher der drei Taster zuletzt gedrückt oder losgelassen wurde, bestimmt den Zustand von Q1.
- **DigitalOutput_Q1**: `logiBUS::io::DQ::logiBUS_QXA`
  - **Parameter**: `QI = TRUE`, `Output = Output_Q1`
  - **Erklärung**: Gibt den Latch-Zustand als physisches Ausgangssignal aus.

Sub-Bausteine: keine

Die Übung verwendet ausschließlich direkte FB-Instanzen, keine SubApp-Instanzen.

Programmablauf und Verbindungen

1. `DigitalInput_I1.IN` ⟶ `AX_ASR_RF_TRIG_1.QI`, `DigitalInput_I2.IN` ⟶ `AX_ASR_RF_TRIG_2.QI`, `DigitalInput_I3.IN` ⟶ `AX_ASR_RF_TRIG_3.QI`: Die drei Taster speisen je einen eigenen Flankendetektor.
2. `AX_ASR_RF_TRIG_1.Q` ⟶ `ASR_MERGE_3.IN1`, `AX_ASR_RF_TRIG_2.Q` ⟶ `ASR_MERGE_3.IN2`, `AX_ASR_RF_TRIG_3.Q` ⟶ `ASR_MERGE_3.IN3`: Alle drei gebündelten ASR-Signale laufen im Merge-Baustein zusammen.
3. `ASR_MERGE_3.OUT` ⟶ `ASR_AX_SR.S_R`: Das zusammengeführte Signal steuert das gemeinsame SR-Latch.
4. `ASR_AX_SR.Q` ⟶ `DigitalOutput_Q1.OUT`: Der Latch-Zustand wird auf den physischen Ausgang `Output_Q1` geschaltet.
5. Ergebnis: Jeder der drei Taster kann `Q1` beliebig oft in beliebiger Reihenfolge ein- und ausschalten — Last-Wins, wie bereits in 229/230 beschrieben. Verglichen mit `Uebung_230_AX` ist das Muster identisch, nur um eine dritte Quelle erweitert.

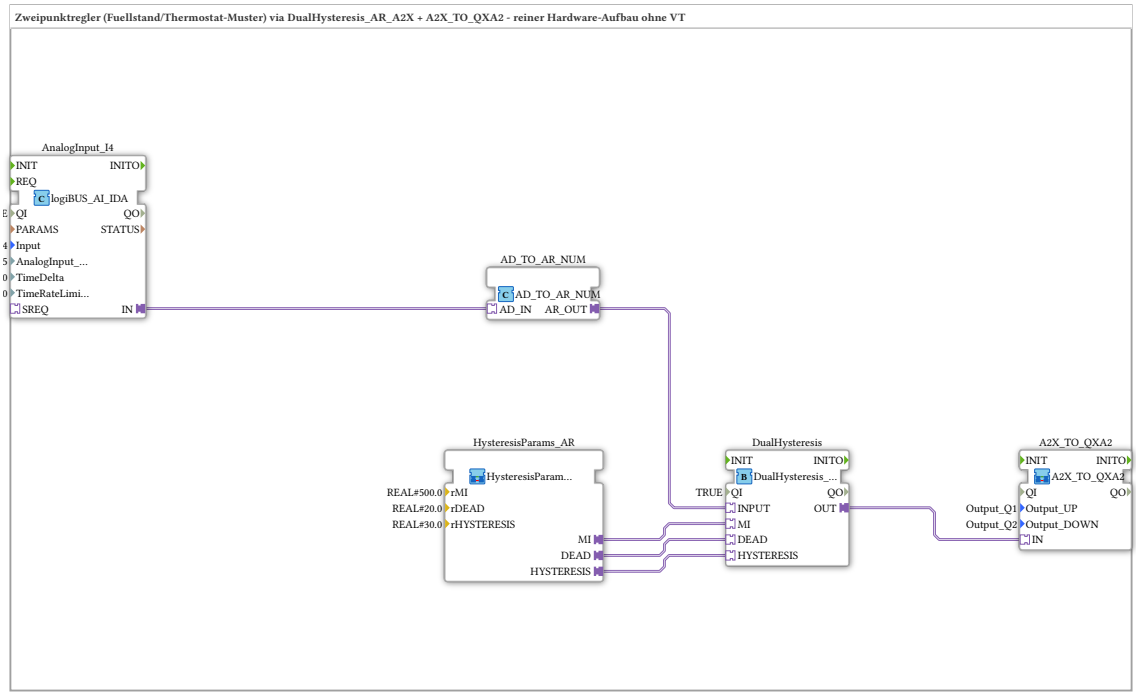
Zusammenfassung

`Uebung_233_AX` demonstriert, dass sich das aus 229/230 bekannte Last-Wins-Latch-Muster durch einfaches Austauschen von `ASR_MERGE_2` gegen `ASR_MERGE_3` (plus einem zusätzlichen `AX_ASR_RF_TRIG`) verlustfrei auf eine dritte unabhängige Quelle erweitern lässt. Die MERGE-Familie (`ASR_MERGE_2` bis `ASR_MERGE_7`) skaliert damit auf mehrere Quellen bis zu sieben Eingänge, ohne die zugrundeliegende ODER-Logik oder das Latch-Verhalten zu verändern — ein wichtiger Baustein für den Bau modularer, gut erweiterbarer Steuerungslogik.

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

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

2.1.2.535 Uebung_234_AX: Zweipunktregler mit Hysterese (reine Hardware-Übung, ohne VT)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung implementiert ein klassisches Thermostat-/Füllstandsschalter-Muster: Ein analoger Messwert (z. B. Füllstand oder Temperatur) wird gegen einen Mittelwert `MI` mit Totzone (`DEAD`) und Hysterese (`HYSTERESIS`) geprüft und steuert darüber zwei entgegengesetzte Aktoren (z. B. Befüllen/Entleeren oder Heizen/Kühlen) an – nie beide gleichzeitig. Die Übung arbeitet ausschließlich mit physischer Hardware, ohne VT-Anzeige.

Verwendete Funktionsbausteine (FBs)

- **AnalogInput_I4**: Analogeingang
  - **Typ**: `logiBUS::io::AI::logiBUS_AI_IDA`
  - **Parameter**: `QI = TRUE`, `Input = AnalogInput_I4`, `AnalogInput_hysteresis = 5`, `TimeDelta = 250`, `TimeRateLimit = 100`
  - **Erklärung**: Liest den rohen Analogmesswert (Adapter `IN`, Typ AD) ein und filtert kleine Signalschwankungen über die eingangsseitige Hysterese von 5.
- **AD_TO_AR_NUM**: Numerische Konvertierung von AD nach AR
  - **Typ**: `adapter::conversion::unidirectional::AD_TO_AR_NUM`
  - **Parameter**: keine
  - **Erklärung**: Wandelt den rohen Messwert (AD) numerisch (AD ⟶ UDINT ⟶ REAL) in einen `REAL`-Adapter (`AR`) um – wie in `Uebung_028a_AR` („ein AD_TO_AR wäre wie ein reinterpret_cast“).
- **HysteresisParams_AR**: Parameter-Baustein für die Hysterese-Schwellwerte (SubApp)
  - **Typ**: `MyLib::sys::HysteresisParams_AR`
  - **Parameter**: `rMI = 500.0`, `rDEAD = 20.0`, `rHYSTERESIS = 30.0`
  - **Erklärung**: Bündelt `MI`, `DEAD` und `HYSTERESIS` als drei feste AR-Konstanten (je ein `initval_AR`); die Werte sind am Aufruf einstellbar. Hier sind es feste Beispielwerte, in echten Anlagen wären sie typischerweise per INI/OPC-UA parametrierbar.
- **DualHysteresis**: Zweipunktregler mit Totzone und Hysterese
  - **Typ**: `logiBUS::signalprocessing::hysteresis::DualHysteresis_AR_A2X`
  - **Parameter**: `QI = TRUE`
  - **Erklärung**: Vergleicht den Messwert (`INPUT`) gegen `MI`: Steigt der Wert über `MI + DEAD + HYSTERESIS` (hier 550), schaltet `UP` ein; fällt er unter `MI - DEAD - HYSTERESIS` (hier 450), schaltet `DOWN` ein.

Ausgeschaltet wird erst wieder, wenn der Wert in die reine Totzone `MI ± DEAD` (480–520) zurückkehrt – die Differenz zwischen Ein- und Ausschaltpunkt verhindert Flattern nahe der Schaltschwelle.

- **A2X_TO_QXA2**: Entbündeln auf zwei Ausgänge (SubApp)
  - **Typ**: `MyLib::sys::A2X_TO_QXA2`
  - **Parameter**: `Output_UP = Output_Q1`, `Output_DOWN = Output_Q2`
  - **Erklärung**: Entbündelt das eine `A2X`-Ausgangssignal (`UP`/`DOWN`) von `DualHysteresis` auf zwei physische logiBUS-Ausgänge (`Q1` für UP, `Q2` für DOWN).

Programmablauf und Verbindungen

1. `AnalogInput_I4.IN` ⟶ `AD_TO_AR_NUM.AD_IN`: Der Rohwert wird numerisch in einen `AR`-Wert (REAL-Adapter) gewandelt.
2. `AD_TO_AR_NUM.AR_OUT` ⟶ `DualHysteresis.INPUT`: Der aufbereitete Messwert gelangt an den Zweipunktregler.
3. `HysteresisParams_AR.MI` / `.DEAD` / `.HYSTERESIS` ⟶ `DualHysteresis.MI` / `.DEAD` / `.HYSTERESIS`: Die Schwellwerte (500.0 / 20.0 / 30.0) werden fest vorgegeben.
4. `DualHysteresis.OUT` ⟶ `A2X_TO_QXA2.IN`: Das kombinierte UP/DOWN-Ergebnis wird an den Ausgabe-Baustein übergeben.
5. `A2X_TO_QXA2` schreibt UP auf `Output_Q1` und DOWN auf `Output_Q2` – die beiden Aktoren werden nie gleichzeitig aktiviert.

Alle Verbindungen sind Adapterverbindungen.

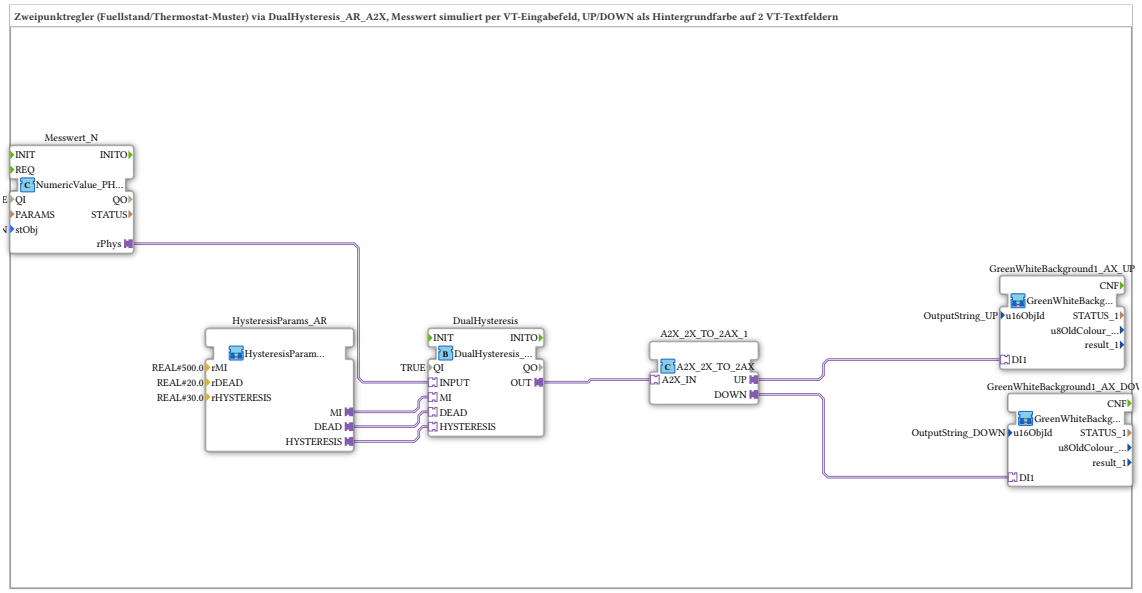
Zusammenfassung

Die Übung 234_AX zeigt den klassischen Aufbau eines Zweipunktreglers mit Hysterese rein auf Hardware-Ebene: Ein Analogeingang wird über den numerischen Adapterkonverter `AD_TO_AR_NUM` aufbereitet, von `DualHysteresis_AR_A2X` gegen einen Mittelwert mit Totzone und Hysterese geprüft und über `A2X_TO_QXA2` auf zwei sich gegenseitig ausschließende Digitalausgänge geschaltet. Für dieselbe Funktion mit VT-Anzeige (Messwert als Zahlenfeld, UP/DOWN als Hintergrundfarbe statt physischer Ausgänge) siehe `Uebung_235_AX`.

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

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

2.1.2.536 Uebung_235_AX: Zweipunktregler mit Hysterese (VT-Variante)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung zeigt dasselbe Zweipunktregler-Muster wie `Uebung_234_AX` (Totzone + Hysterese um einen Mittelwert), simuliert den Messwert jedoch über ein VT-Eingabefeld statt über einen echten Analogsensor und meldet den Reglerzustand über die Hintergrundfarbe zweier VT-Textfelder zurück statt über physische Ausgänge.

Verwendete Funktionsbausteine (FBs)

- **Messwert_N**: Terminal-Eingabe (Typ: `isobus::UT::io::NumericValue::NumericValue_PHYSA`)
  - **Parameter**: QI = TRUE, stObj = InputNumber_Messwert_N
  - **Erklärung**: Liest bei jeder Änderung von `InputNumber_Messwert` (VT-Objekt 9002, Bereich 0–1000) den physikalischen Wert als AR-Plug (`rPhys`) ein. Der Bediener simuliert damit im Pool `Workspace_Dreieck` den Analogsensor aus `Uebung_234_AX`.
- **HysteresisParams_AR**: Parameter-Baustein für die Hysterese-Schwellwerte (SubApp-Instanz, Typ: `MyLib::sys::HysteresisParams_AR`)
  - **Parameter**: rMI = 500.0, rDEAD = 20.0, rHYSTERESIS = 30.0
  - **Erklärung**: Stellt die drei Schwellwerte des Zweipunktreglers bereit – dieselben Werte wie in `Uebung_234_AX`.
- **DualHysteresis**: Zweipunktregler mit Totzone und Hysterese (Typ: `logiBUS::signalprocessing::hysteresis::DualHysteresis_AR_A2X`)
  - **Parameter**: QI = TRUE
  - **Erklärung**: Vergleicht den Messwert gegen `MI`: Steigt der Wert über `MI + DEAD + HYSTERESIS` (550), schaltet `UP`; fällt er unter `MI - DEAD - HYSTERESIS` (450), schaltet `DOWN`. Ausgeschaltet wird erst innerhalb der reinen Totzone `MI ± DEAD` (480–520) – die Differenz zwischen Ein- und Ausschaltpunkt verhindert Flattern.
- **A2X_2X_TO_2AX_1**: Entbündelung eines A2X-Signals in zwei AX-Signale (Typ: `adapter::conversion::unidirectional::A2X_2X_TO_2AX`)
  - **Parameter**: keine
  - **Erklärung**: `DualHysteresis.OUT` liefert UP/DOWN gebündelt als ein `A2X`-Signal. Dieser Baustein entbündelt es wieder in zwei einzelne `AX`-Signale, weil `GreenWhiteBackground1_AX` pro Instanz nur einen `DI1`-Eingang besitzt.

Sub-Bausteine: GreenWhiteBackground1_AX_UP, GreenWhiteBackground1_AX_DOWN

- **GreenWhiteBackground1_AX_UP** (Typ: `MyLib::sys::GreenWhiteBackground1_AX`)
  - **Parameter**: u16ObjId = OutputString_UP
  - **Erklärung**: Färbt das VT-Textfeld `OutputString_UP` (VT-Objekt 11001) grün, solange `UP` aktiv ist, sonst weiß.
- **GreenWhiteBackground1_AX_DOWN** (Typ: `MyLib::sys::GreenWhiteBackground1_AX`)
  - **Parameter**: u16ObjId = OutputString_DOWN
  - **Erklärung**: Färbt das VT-Textfeld `OutputString_DOWN` (VT-Objekt 11002) grün, solange `DOWN` aktiv ist, sonst weiß.

Programmablauf und Verbindungen

1. Ändert der Bediener `InputNumber_Messwert`, liefert `Messwert_N.rPhys` den neuen Wert.
2. `DualHysteresis.INPUT` ← `Messwert_N.rPhys`: Der simulierte Messwert gelangt an den Zweipunktregler.
3. `DualHysteresis.MI/DEAD/HYSTERESIS` ← `HysteresisParams_AR.MI/DEAD/HYSTERESIS`: Die Schwellwerte (500.0 / 20.0 / 30.0) werden fest vorgegeben.
4. `A2X_2X_TO_2AX_1.A2X_IN` ← `DualHysteresis.OUT`: Das kombinierte UP/DOWN-Signal wird entbündelt.
5. `GreenWhiteBackground1_AX_UP.DI1` ← `A2X_2X_TO_2AX_1.UP`: Der UP-Zustand färbt `OutputString_UP`.
6. `GreenWhiteBackground1_AX_DOWN.DI1` ← `A2X_2X_TO_2AX_1.DOWN`: Der DOWN-Zustand färbt `OutputString_DOWN`.
7. Am echten Terminal: Messwert deutlich über 550 → `OutputString_UP` grün, `OutputString_DOWN` bleibt weiß. Messwert deutlich unter 450 → umgekehrt. Messwert zwischen 480 und 520 → beide weiß. Werte zwischen 520–550 bzw. 450–480 (innerhalb der Hysterese, außerhalb der Totzone) zeigen das Hysterese-Verhalten: Der zuletzt aktive Zustand bleibt erhalten, bis die jeweilige Totzonen-Grenze erreicht wird.

Zusammenfassung

Die Übung 235_AX ist das VT-Pendant zu `Uebung_234_AX`: Anstelle eines echten Analogsensors simuliert der Bediener den Messwert über ein VT-Eingabefeld, und anstelle physischer Ausgänge zeigt die Hintergrundfarbe zweier VT-Textfelder den UP-/DOWN-Zustand des Zweipunktreglers an. Die eigentliche Regellogik (`DualHysteresis_AR_A2X` mit Totzone und Hysterese um einen Mittelwert) ist in beiden Übungen identisch. Die dafür neu angelegten VT-Objekte – `InputNumber_Messwert` (9002) mit der zugehörigen Zahlenvariable `NumberVariable_Messwert` (21002) sowie `OutputString_UP` (11001) und `OutputString_DOWN` (11002) – liegen in `Workspace_Dreieck/DefaultPool/DefaultPool.jop`.

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

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

2.1.2.537 Übung 235: Zweipunktregler mit Hysterese (VT-Variante)

Thema: `DualHysteresis_AR_A2X` — Totzone + Hysterese um einen Mittelwert, Messwert und Rückmeldung komplett auf dem VT

Situationsbeschreibung

Klassisches Thermostat-/Füllstandsschalter-Muster: ein analoger Messwert wird gegen einen Mittelwert `MI` mit Totzone (`DEAD`) und Hysterese (`HYSTERESIS`) geprüft und schaltet darüber zwei entgegengesetzte Aktoren (z.B. Heizen/Kühlen oder Befüllen/Entleeren) — nie beide gleichzeitig. Im Pool `Workspace_Dreieck` simuliert der Bediener den Messwert über `InputNumber_Messwert` (VT-Objekt 9002, Bereich 0–1000) statt über einen echten Analogsensor; `OutputString_UP`/`OutputString_DOWN` (VT-Objekte 11001/11002) zeigen per Hintergrundfarbe an, welcher der beiden Aktoren gerade aktiv wäre.

Funktionsbeschreibung

- **Messwert lesen:** `Messwert_N` (`NumericValue_PHYSA`, `stObj := InputNumber_Messwert_N`) liefert bei jeder Änderung von `InputNumber_Messwert` den physikalischen Wert bereits direkt als `AR`-Adapter-Plug (`rPhys`) — anders als die reine Event-/Datenvariante `NumericValue_PHYS` braucht es hier keine separate Umwandlung in einen Adapter, `rPhys` wird direkt an `DualHysteresis_AR_A2X.INPUT` angeschlossen.
- **Regeln:** `DualHysteresis_AR_A2X` vergleicht den Messwert gegen `MI` (hier fest 500.0 über `HysteresisParams_AR`, im echten Einsatz parametrierbar): steigt der Wert über `MI + DEAD + HYSTERESIS` (hier 550), schaltet `UP`; fällt er unter `MI - DEAD - HYSTERESIS` (hier 450), schaltet `DOWN`. Ausgeschaltet wird erst innerhalb der reinen Totzone `MI ± DEAD` (480–520) — die Differenz zwischen Ein- und Ausschaltpunkt verhindert Flattern.
- **Anzeigen:** `DualHysteresis.OUT` liefert `UP/DOWN` gebündelt als ein `A2X`-Signal. `A2X_2X_TO_2AX` entbündelt es wieder in zwei einzelne `AX`-Signale, weil `GreenWhiteBackground1_AX` pro Instanz nur einen `DI1`-Eingang hat — je eine Instanz färbt `OutputString_UP` bzw. `OutputString_DOWN` grün/weiß.

Arbeitsauftrag

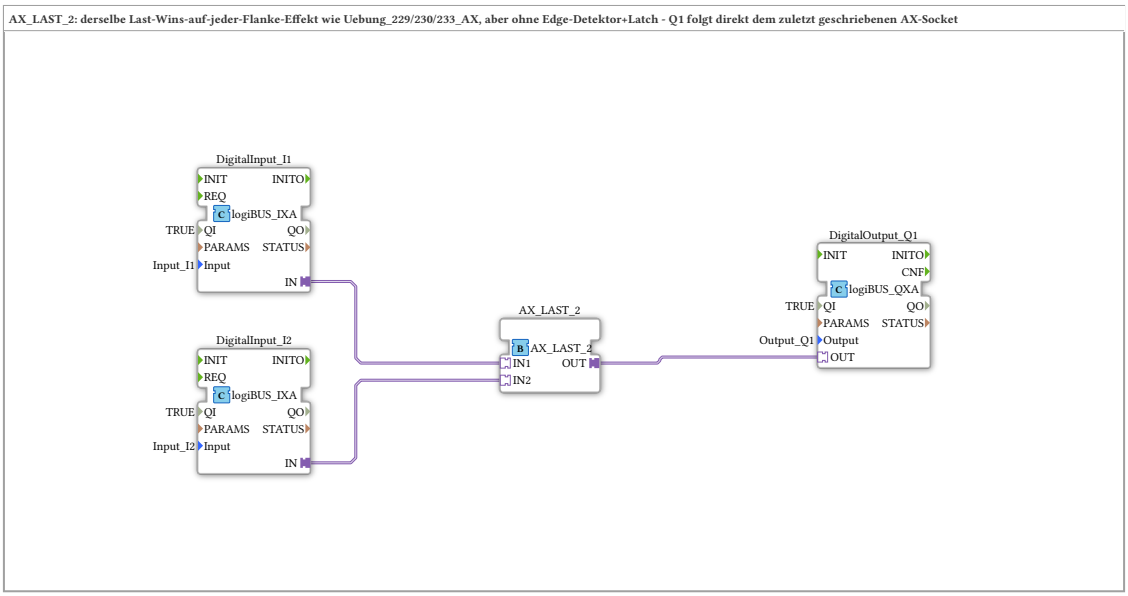
1. Legen Sie die SubApp `Uebung_235_AX` an (bereits als Referenzlösung vorhanden).
2. Lesen Sie `InputNumber_Messwert` (VT-Objekt 9002) mit einer `NumericValue_PHYSA`-Instanz (`Messwert_N`, `stObj := InputNumber_Messwert_N` aus `DefaultPool_Dreieck_Numeric.gcf`) — sie liefert den Wert bereits direkt als `AR`-Adapter-Plug (`rPhys`).
3. Verbinden Sie `Messwert_N.rPhys` direkt mit `DualHysteresis_AR_A2X.INPUT` (keine separate Umwandlung nötig). `MI`/`DEAD`/`HYSTERESIS` über eine `HysteresisParams_AR`-Instanz setzen (Standardwerte 500.0/20.0/30.0).
4. Entbündeln Sie `DualHysteresis.OUT` mit `A2X_2X_TO_2AX` und speisen Sie `UP/DOWN` in je eine `GreenWhiteBackground1_AX`-Instanz (`u16ObjId := OutputString_UP` bzw. `OutputString_DOWN` aus `DefaultPool_Dreieck.gcf`).
5. Am echten Terminal testen: `InputNumber_Messwert` auf einen Wert deutlich über 550 setzen → `OutputString_UP` wird grün, `OutputString_DOWN` bleibt weiß. Wert deutlich unter 450 → umgekehrt. Wert zwischen 480 und 520 → beide weiß. Werte zwischen 520–550 bzw. 450–480 (innerhalb der Hysterese, außerhalb der Totzone) zeigen das Hysterese-Verhalten: der zuletzt aktive Zustand bleibt erhalten, bis die jeweilige Totzonen-Grenze erreicht wird.

Referenzlösung

`Uebung_235_AX.SUB`. Für die reine Hardware-Variante (Analogsensor statt VT-Eingabe, physische Ausgänge statt Hintergrundfarbe) siehe `Uebung_234_AX`. Die neuen VT-Objekte (`InputNumber_Messwert`

9002, `NumberVariable_Messwert` 21002, `OutputString_UP` 11001, `OutputString_DOWN` 11002) liegen in `Ventilsteuerung\ISO-DesignerProjects\Workspace_Dreieck\DefaultPool\DefaultPool.jop`, platziert auf `DataMask_M1` unterhalb der bestehenden Sollwert-/Istwert-Felder; die zugehörigen Konstanten in `DefaultPool_Dreieck.gcf/DefaultPool_Dreieck_Numeric.gcf`.

2.1.2.538 Uebung_236_AX: AX_LAST_2 – derselbe Last-Wins-Effekt wie 229/230/233, ohne Edge-Detektor und Latch



• – ◦ • – ◦ • – ◦ •

Einleitung

Zwei Taster `I1`/`I2` steuern gemeinsam einen Ausgang `Q1` – dieselbe Grundidee wie in `Uebung_229_AX`/`Uebung_230_AX`/`Uebung_233_AX`, aber mit einem völlig anderen Baustein: `AX_LAST_2` ersetzt die komplette Kette aus Flankenerkennung, Zusammenführung und Latch durch einen einzigen Adapter-Baustein. **Wichtig:** `Uebung_236_AX` verhält sich beobachtbar **identisch** zu `Uebung_229_AX`/`Uebung_230_AX`/`Uebung_233_AX` – der Unterschied liegt im Bauplan, nicht im Verhalten.

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_I1**, **DigitalInput_I2**: logiBUS Digitaleingänge (Typ: `logiBUS::io::DI::logiBUS_IXA`)
  - **Parameter**: QI = TRUE, Input = Input_I1 bzw. Input_I2
  - **Erklärung**: Wandeln die physischen Tasterzustände in AX-Adaptersignale um. Der Adapter-Schreibevent (`IN.E1`) feuert dabei nur bei echter Zustandsänderung (über `logiBUS_IX.IND`) – er ist also selbst schon „flankengetriggert“, genau wie `E_RF_TRIG` in `AX_RF_TRIG`/`AX_ASR_RF_TRIG`.
- **AX_LAST_2**: „Wer zuletzt schrieb, gewinnt“-Baustein (Typ: `adapter::events::unidirectional::AX_LAST_2`)
  - **Parameter**: keine
  - **Erklärung**: Besitzt zwei AX-Sockets (`IN1`, `IN2`) und einen AX-Plug (`OUT`). Sein ECC ist denkbar einfach: Welcher Socket zuletzt sein eigenes Schreibevent (`IN1.E1`/`IN2.E1`) ausgelöst hat, dessen aktueller Datenwert wird sofort 1:1 an `OUT` durchgereicht.
- **DigitalOutput_Q1**: logiBUS Digitalausgang (Typ: `logiBUS::io::DQ::logiBUS_QXA`)
  - **Parameter**: QI = TRUE, Output = Output_Q1
  - **Erklärung**: Gibt den aktuellen Zustand von `AX_LAST_2.OUT` an die Peripherie weiter.

Sub-Bausteine: keine

Die Übung verwendet keine weiteren Unterbausteine, alle FBs sind direkt auf der obersten Ebene der Sub-App angeordnet – `DigitalInput_I1`/`_I2` sind ohne jeden Zwischenbaustein direkt mit `AX_LAST_2.IN1`/`IN2` verbunden.

Programmablauf und Verbindungen

1. `AX_LAST_2.IN1` ⟵ `DigitalInput_I1.IN`: Der Zustand von Taster I1 wird direkt an den ersten Socket von `AX_LAST_2` angeschlossen.
2. `AX_LAST_2.IN2` ⟵ `DigitalInput_I2.IN`: Der Zustand von Taster I2 wird direkt an den zweiten Socket angeschlossen – kein Flankendetektor, kein Merge-Baustein, kein Latch dazwischen.
3. `DigitalOutput_Q1.OUT` ⟵ `AX_LAST_2.OUT`: `Q1` folgt jeweils dem Wert desjenigen Sockets, der zuletzt ein Schreibevent ausgelöst hat.
4. **Testablauf**: `I1` drücken ⟶ `Q1` EIN. `I2` zusätzlich drücken und halten ⟶ `Q1` bleibt EIN (I2 hat zuletzt geschrieben). `I2` loslassen (I1 bleibt gedrückt) ⟶ `Q1` fällt sofort auf AUS – identisch zum Verhalten von `Uebung_229_AX`/`Uebung_230_AX`, nur mit einem einzigen Baustein statt drei.

Warum das identisch zu 229/230/233 ist, nicht anders: `AX_RF_TRIG`/`AX_ASR_RF_TRIG` meldet die steigende Flanke (Drücken) als SET und die fallende Flanke (Loslassen) als RESET auf denselben Latch – jedes Loslassen irgendeines Tasters schaltet `Q1` sofort AUS, selbst wenn ein anderer Taster noch gedrückt gehalten wird. Das ist bereits dieselbe „letzte Flanke gewinnt"-Logik wie bei `AX_LAST_2`, kein Speicher, der sich merkt, dass ein anderer Taster noch aktiv ist. Der eigentliche Unterschied zwischen den beiden Bauplänen liegt in der Konstruktion, nicht im Verhalten:

- `Uebung_229_AX`: 2× `AX_RF_TRIG` + gemeinsamer `AX_SR` (lose EventConnections).
- `Uebung_230_AX`: 2× `AX_ASR_RF_TRIG` + `ASR_MERGE_2` + `ASR_AX_SR` (dieselbe Funktion, als Adapter-Bausteine).
- `Uebung_236_AX`: **ein einziger** Baustein (`AX_LAST_2`) ersetzt die komplette Kette aus Flankenerkennung, Zusammenführung und Latch, weil er direkt auf den rohen AX-Schreibereignissen arbeitet statt auf explizit erzeugten SET/RESET-Kommandos.

Skalierbarkeit gegenüber `Uebung_233_AX` (`ASR_MERGE_3`): `ASR_MERGE_N` existiert als ganze Familie (`ASR_MERGE_2..7`) – von 2 auf 3 Quellen zu gehen (230 ⟶ 233) heißt nur: einen weiteren `AX_ASR_RF_TRIG` anschließen und `ASR_MERGE_2` durch `ASR_MERGE_3` ersetzen, alles in einem einzigen Baustein. `AX_LAST_2` hat dagegen **keine** höherwertige Geschwister – es gibt kein `AX_LAST_3` in der Bibliothek. Eine dritte Rohquelle lässt sich nur durch Kaskadieren zweier `AX_LAST_2`-Instanzen einbinden (`OUT` der einen auf `IN1`/`IN2` der nächsten). Das bleibt aber semantisch korrektes Last-Wins über alle drei Quellen: `AX_LAST_2` kopiert den gewählten Wert nach `OUT.D1` **und** löst `OUT.E1` aus, jede Änderung an einer der ursprünglichen Quellen läuft also als frisches Ereignis durch die Kaskade – anders als bei einer reinen ODER-Verknüpfung geht dabei keine Aktualität verloren. Der Unterschied zu `ASR_MERGE_N` ist rein baulich: eine Familie mit einem Baustein pro Eingangszahl vs. N-1 kaskadierte Instanzen desselben Zwei-Eingänge-Bausteins – keine unterschiedliche Semantik.

Zusammenfassung

Die Übung 236_AX zeigt, dass sich die aus 229/230/233 bekannte „Last-Wins"-Verriegelung von zwei Tastern auf einen gemeinsamen Ausgang mit einem einzigen Baustein (`AX_LAST_2`) nachbilden lässt, ohne einen expliziten Flankendetektor, Merge-Baustein oder Latch zu benötigen – weil `AX_LAST_2` direkt auf den ohnehin schon flankengetriggerten AX-Schreibereignissen arbeitet. Das beobachtbare Verhalten ist zu den drei Vorgängerübungen identisch; der Unterschied liegt ausschließlich in der Anzahl und Art der verwendeten Bausteine, sowie darin, dass `AX_LAST_2` – anders als die `ASR_MERGE_N`-Familie – nicht direkt auf mehr als zwei Quellen skaliert.

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

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

2.1.2.539 Übung 236: Derselbe Last-Wins-Effekt wie 229/230/233 — ohne Edge-Detektor und Latch

Thema: `AX_LAST_2` — gleiches Verhalten, drastisch weniger Bausteine

Situationsbeschreibung

Zwei Taster `I1`/`I2` steuern gemeinsam einen Ausgang `Q1` — dieselbe Grundidee wie in `Uebung_229_AX`/`Uebung_230_AX`, aber mit einem völlig anderen Baustein: `AX_LAST_2` (`adapter::events::unidirectional`) statt `AX_RF_TRIG`/`AX_ASR_RF_TRIG`+`ASR_MERGE_2`+`AX_SR`/`ASR_AX_SR`.

Wichtig, korrigiert gegenüber einer früheren Fassung dieser Beschreibung: `Uebung_236_AX` verhält sich **beobachtbar identisch** zu `Uebung_229_AX`/`Uebung_230_AX`/`Uebung_233_AX` — nicht anders, wie hier ursprünglich fälschlich behauptet wurde. Der Unterschied liegt im Bauplan, nicht im Verhalten.

Warum 229/230/233 KEIN „hält nach Loslassen"-Latch sind

`AX_RF_TRIG`/`AX_ASR_RF_TRIG` meldet die steigende Flanke (Drücken) als `SET` UND die fallende Flanke (Loslassen) als `RESET` — beide Flanken BEIDER (bzw. aller drei) Taster wirken auf denselben `AX_SR`/`ASR_AX_SR`. Das bedeutet: **jedes** Loslassen irgendeines Tasters schaltet `Q1` sofort AUS, selbst wenn ein anderer Taster noch gedrückt gehalten wird — `Uebung_229_AX`s eigene Beschreibung sagt das bereits explizit: „Der Ausgang folgt also nicht dem Tasterzustand, sondern immer der zuletzt aufgetretenen Flanke (Drücken oder Loslassen, an welchem Taster auch immer)." Das ist also kein Speicher, der „merkt", dass ein anderer Taster noch aktiv ist — es ist bereits dieselbe „letzte Flanke gewinnt"-Logik wie bei `AX_LAST_2`.

Funktionsbeschreibung

- **`AX_LAST_2`** hat zwei AX-Sockets (`IN1`, `IN2`) und einen AX-Plug (`OUT`). Sein ECC ist denkbar einfach: welcher Socket zuletzt sein eigenes Event (`IN1.E1`/`IN2.E1`) ausgelöst hat, dessen aktueller Datenwert (`D1`) wird sofort 1:1 an `OUT` durchgereicht.
- **Direkt verdrahtet**: `DigitalInput_I1`/`_I2` (`logiBUS_IXA`) gehen ohne Umweg über `AX_LAST_2.IN1`/`IN2` — `logiBUS_IXA.IN.E1` feuert (über `logiBUS_IX.IND`) ohnehin nur bei echter Zustandsänderung, ist also selbst schon „edge-getriggert", genau wie `E_RF_TRIG` in `AX_(ASR_)RF_TRIG`. Deshalb entsteht hier ohne jeden Zwischenbaustein exakt dieselbe „letzte Flanke gewinnt"-Sequenz wie in 229/230/233.
- **Der eigentliche Unterschied zu 229/230/233 ist der Bauplan, nicht das Verhalten:**
  - `Uebung_229_AX`: 2 `AX_RF_TRIG` + gemeinsamer `AX_SR` (lose EventConnections).
  - `Uebung_230_AX`: 2 `AX_ASR_RF_TRIG` + `ASR_MERGE_2` + `ASR_AX_SR` (dieselbe Funktion, als Adapter-Bausteine statt loser Events).
  - `Uebung_236_AX`: **ein einziger** Baustein (`AX_LAST_2`) ersetzt die komplette Kette aus Flankenerkennung + Merge + Latch, weil er direkt auf den rohen AX-Schreibereignissen arbeitet statt auf explizit erzeugten SET/RESET-Kommandos.

Unterschied zu Übung 233 (`ASR_MERGE_3`) — Skalierbarkeit der Baustein-Familie

Der einzige *substantielle* Unterschied zwischen dieser Familie und `AX_LAST_2` ist nicht das Verhalten, sondern wie gut sie auf mehr als 2 Quellen skalieren:

- **`ASR_MERGE_N`** existiert als ganze Familie (`ASR_MERGE_2..7`) — von 2 auf 3 Quellen zu gehen (Übung 230 → 233) heißt nur: einen weiteren `AX_ASR_RF_TRIG` anschließen und `ASR_MERGE_2` durch `ASR_MERGE_3` ersetzen (ein zusätzlicher Socket `IN3`, sonst identisch). Eine vierte Quelle wäre genauso einfach `ASR_MERGE_4`.
- **`AX_LAST_2`** hat dagegen **keine** höherwertige Geschwister — es gibt kein `AX_LAST_3` in der Bibliothek. Eine dritte Rohquelle lässt sich nur durch **Kaskadieren** zweier `AX_LAST_2`-Instanzen einbinden (`OUT` der einen auf `IN1`/`IN2` der nächsten). Das bleibt aber semantisch korrektes Last-Wins über alle drei Quellen, keine zweistufige Rangfolge: `AX_LAST_2` kopiert bei jeder Übernahme sowohl den Wert nach `OUT.D1` als auch löst es `OUT.E1` aus, jede Änderung an einer der ursprünglichen Quellen läuft also bei jedem Kaskadenschritt als frisches Event durch. Der Unterschied zu `ASR_MERGE_N` ist rein baulich (1 Baustein pro Eingangszahl vs. N-1 kaskadierte Instanzen), keine andere Semantik. Wie bei jeder IEC-61499-Anwendung mit mehreren unabhängigen Event-Quellen garantiert die Norm allerdings keine feste Reihenfolge für nahezu gleichzeitig eintreffende Events an verschiedenen Sockets (`IN1`/`IN2`) — „wer zuletzt schrieb" bezieht sich auf die tatsächliche Verarbeitungsreihenfolge im Laufzeitsystem, nicht auf einen exakten Quell-Zeitstempel.

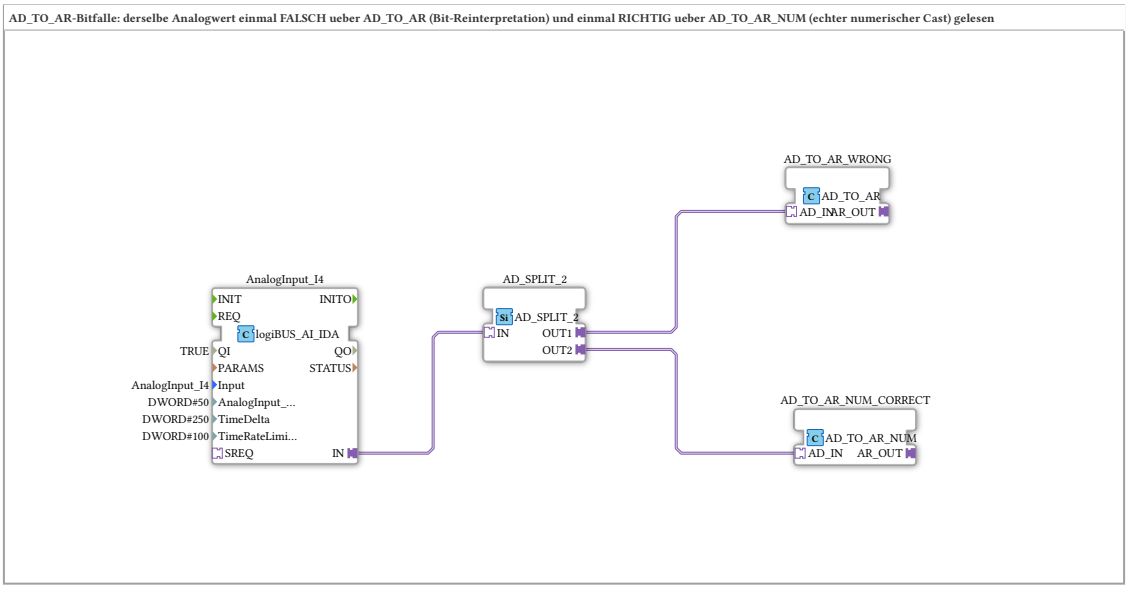
Arbeitsauftrag

1. Legen Sie die SubApp `Uebung_236_AX` an (bereits als Referenzlösung vorhanden).
2. Verbinden Sie `Input_I1`/`Input_I2` direkt (ohne Zwischenbaustein) mit `AX_LAST_2.IN1`/`IN2`.
3. Verbinden Sie `AX_LAST_2.OUT` mit `Output_Q1`.
4. Am echten Terminal/Hardware testen und mit Übung 229/230 vergleichen: `I1` drücken → `Q1` ON; `I2` zusätzlich drücken und halten → `Q1` bleibt ON (I2 hat zuletzt geschrieben); `I2` loslassen (`I1` bleibt gedrückt) → `Q1` fällt sofort auf OFF — identisch zum Verhalten von Übung 229/230, nur mit einem einzigen Baustein statt drei.
5. Vergleichen Sie die beiden SubApps im 4diac-Editor: `Uebung_230_AX` hat 4 Bausteine (2x `AX_ASR_RF_TRIG`, `ASR_MERGE_2`, `ASR_AX_SR`) zwischen Ein- und Ausgang, `Uebung_236_AX` nur einen einzigen (`AX_LAST_2`) — für exakt dasselbe Verhalten.

Referenzlösung

`Uebung_236_AX.SUB`. Der verwendete Baustein liegt in `Ventilsteuerung\4diacIDE-workspace\.lib\adapter-3.0.0\typelib\events\unidirectional\BOOL\AX_LAST_2.fbt`.

2.1.2.540 Uebung_237_AX: AD_TO_AR-Bitfalle – Bit-Reinterpretation vs. numerischer Cast



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung ist bewusst als Vergleich zwischen einer **falschen** und einer **richtigen** Lösung aufgebaut: Derselbe rohe Analogwert wird auf zwei Arten von `AD` (DWORD-Adapter) nach `AR` (REAL-Adapter) konvertiert – einmal fälschlich über `AD_TO_AR` (Bit-Reinterpretation) und einmal korrekt über `AD_TO_AR_NUM` (echter numerischer Cast). Die Instanznamen `AD_TO_AR_WRONG` und `AD_TO_AR_NUM_CORRECT` sind bewusst so gewählt, um die Fallstricke bei der Konvertierung von Analogwerten sichtbar zu machen.

Verwendete Funktionsbausteine (FBs)

- **AnalogInput_I4**: Analogeingang (Typ: `logiBUS::io::AI::logiBUS_AI_IDA`)
  - **Parameter**: QI = TRUE, Input = AnalogInput_I4, AnalogInput_hysteresis = 50, TimeDelta = 250, TimeRateLimit = 100
  - **Erklärung**: Liest den rohen Analogmesswert als `AD`-Adapter (DWORD) ein.
- **AD_SPLIT_2**: Verteilung eines AD-Signals auf zwei Ziele (Typ: `adapter::events::unidirectional::AD_SPLIT_2`)
  - **Parameter**: keine
  - **Erklärung**: Dupliziert den einen rohen Analogwert verlustfrei, damit er parallel über beide Konvertierungswege beobachtet werden kann.
- **AD_TO_AR_WRONG**: Konvertierung von AD nach AR – FALSCH (Typ: `adapter::conversion::unidirectional::AD_TO_AR`)
  - **Parameter**: keine
  - **Erklärung**: Sieht wie die naheliegende direkte Konvertierung aus, ist intern aber `F_DWORD_TO_REAL` – eine reine IEEE754-Bit-Reinterpretation. Ein roher Analogwert wie `DWORD#2048` wird dadurch **nicht** zu `REAL#2048.0`, sondern zu einer bedeutungslosen Zahl nahe Null.
- **AD_TO_AR_NUM_CORRECT**: Konvertierung von AD nach AR – RICHTIG (Typ: `adapter::conversion::unidirectional::AD_TO_AR_NUM`)
  - **Parameter**: keine
  - **Erklärung**: Macht denselben Job numerisch korrekt, über die Zwischenkette DWORD ⟶ UDINT ⟶ REAL – dieselbe Kette, die `Uebung_028a_AR` von Hand aus `AD_TO_AUDI` + `AUDI_TO_AR` zusammensetzt. `AD_TO_AR_NUM` ist das drop-in-kompatible Einzelbaustein-Äquivalent.

Sub-Bausteine: keine

Die Übung verwendet keine weiteren Unterbausteine, alle FBs sind direkt auf der obersten Ebene der SubApp angeordnet.

Programmablauf und Verbindungen

1. `AD_SPLIT_2.IN` ⟵ `AnalogInput_I4.IN`: Der rohe Analogwert wird auf zwei Verbraucher verteilt.
2. `AD_TO_AR_WRONG.AD_IN` ⟵ `AD_SPLIT_2.OUT1`: Der erste Zweig konvertiert per Bit-Reinterpretation (`F_DWORD_TO_REAL`) – das Ergebnis ist eine bedeutungslose Zahl nahe Null.
3. `AD_TO_AR_NUM_CORRECT.AD_IN` ⟵ `AD_SPLIT_2.OUT2`: Der zweite Zweig konvertiert numerisch korrekt (DWORD ⟶ UDINT ⟶ REAL) – das Ergebnis entspricht dem tatsächlichen Messwert.
4. **Arbeitsauftrag**: Beide `AR_OUT.D1`-Pins im laufenden 4diac-Monitor beobachten (rechte Maustaste auf den Pin ⟶ Watch) und vergleichen: `AD_TO_AR_WRONG` zeigt eine kleine, mit dem tatsächlichen Analogwert scheinbar unzusammenhängende Zahl; `AD_TO_AR_NUM_CORRECT` zeigt den plausiblen, tatsächlichen Messwert.

Wann ist `AD_TO_AR` trotzdem richtig? Wenn `AD_IN` bereits ein Bitmuster ist, das als REAL gemeint war (z. B. das Ergebnis von `F_REAL_TO_DWORD` an anderer Stelle) – dann ist die Bit-Reinterpretation genau die richtige, verlustfreie Rückrichtung. Ein roher Analog-/Zählerwert wie in dieser Übung ist das Gegenbeispiel: Hier ist der numerische Cast (`AD_TO_AR_NUM`) richtig.

Zusammenfassung

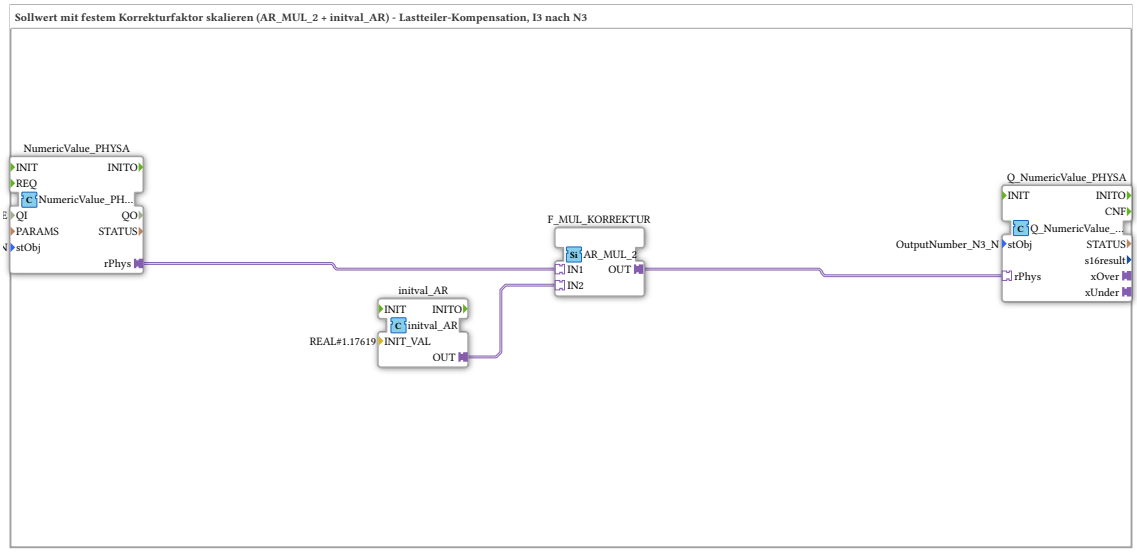
Die Übung 237_AX macht anhand zweier parallel verdrahteter Konvertierungswege sichtbar, dass `AD_TO_AR` und `AD_TO_AR_NUM` grundverschiedene Operationen sind: `AD_TO_AR` ist eine reine Bit-Reinterpretation (richtig nur, wenn der Eingang bereits ein absichtlich so kodiertes REAL-Bitmuster ist), `AD_TO_AR_NUM` ist der numerisch korrekte Cast für rohe Analog- oder Zählerwerte. Wer bei der Konvertierung eines Analogwerts den vermeintlich „direkten“ Baustein `AD_TO_AR` wählt, erhält ein plausibel aussehendes, aber falsches Ergebnis – die Übung schult den bewussten Umgang mit dieser Verwechslungsgefahr.

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

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

2.1.2.541 Uebung_245_AX: Sollwert mit festem Korrekturfaktor skalieren (AR_MUL_2 + initval_AR) - Lastteiler-Kompensation, I3 nach N3

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_245_AX (Sollwert mit festem Korrekturfaktor skalieren (AR_MUL_2 + initval_AR) - Lastteiler-Kompensation, I3 nach N3).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Sollwert mit festem Korrekturfaktor skalieren (AR_MUL_2 + initval_AR) - Lastteiler-Kompensation, I3 nach N3**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_245_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **NumericValue_PHYSA**: Instanz des Typs isobus::UT::io::NumericValue::NumericValue_PHYSA.
  - Parameter QI = TRUE
  - Parameter stObj = InputNumber_I3_N
- **F_MUL_KORREKTUR**: Instanz des Typs adapter::iec61131::arithmetic::AR_MUL_2.
- **Q_NumericValue_PHYSA**: Instanz des Typs isobus::UT::Q::Q_NumericValue_PHYSA.
  - Parameter stObj = OutputNumber_N3_N
- **initval_AR**: Instanz des Typs adapter::types::unidirectional::AR::initval::initval_AR.
  - Parameter INIT_VAL = REAL#1.17619

Verbindungen und Schnittstellen

Adapterverbindungen:

- F_MUL_KORREKTUR.OUT -> Q_NumericValue_PHYSA.rPhys
- initval_AR.OUT -> F_MUL_KORREKTUR.IN2
- NumericValue_PHYSA.rPhys -> F_MUL_KORREKTUR.IN1

Hinweise aus dem Modell

> Korrekturfaktor 1,17619 = 1 / 0,8502: kompensiert einen Lastteiler (z.B. PWM-Tiefpassfilter, dessen Ausgangsimpedanz gegen die Eingangsimpedanz des angeschlossenen Verbrauchers belastet wird - siehe PVEA-Kompensationsrechnung). I3 = unkorrigierter Sollwert 0-100%, N3 = korrigierter Wert, der am unbelasteten Ausgang eingestellt werden muss, damit am Verbraucher der ursprüngliche Sollwert ankommt.

Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_245_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.542 Übung 245: Sollwert mit festem Korrekturfaktor skalieren — AR_MUL_2 zwischen zwei PHYSA-Zahlenfeldern

Thema: Feste lineare Korrektur eines REAL-Sollwerts, rein über AR-Adapter (kein REQ/CNF nötig)

Situationsbeschreibung

Ein Sollwert (z.B. ein Tastgrad in Prozent) durchläuft auf dem Weg zum Verbraucher eine Signalkette mit einem systematischen, aber bekannten und konstanten Fehler — z.B. einen Tiefpassfilter, dessen Ausgangsimpedanz gegen die Eingangsimpedanz des angeschlossenen Verbrauchers einen Lastteiler bildet (Spannungsteiler-Effekt: der Verbraucher „zieht“ das Signal herunter). Wenn dieser Effekt konstant ist (gleiche Bauteile, gleiche Impedanzen), kann er mit einem festen multiplikativen Korrekturfaktor kompensiert werden: der gesendete Sollwert wird um genau den Kehrwert des Lastteiler-Verhältnisses erhöht, damit am Verbraucher der ursprünglich gewünschte Wert ankommt.

Diese Übung baut das Kernstück dieser Kompensation nach: ein Eingabe-Zahlenfeld (I3) liefert den unkorrigierten Sollwert, ein AR_MUL_2 mit einem über initval_AR fest eingespeisten zweiten Faktor skaliert ihn, ein Ausgabe-Zahlenfeld (N3) zeigt das Ergebnis.

Funktionsbeschreibung

- **Eingabe:** NumericValue_PHYSA liest das VT-Eingabefeld InputNumber_I3_N und liefert den physikalischen (bereits skalierten) REAL-Wert über den rPhys-AR-Adapter — siehe Uebung_011f_PHYSA für das reine Durchschleifen ohne Umrechnung.
- **Fester Faktor als AR-Konstante:** initval_AR (INIT_VAL = REAL#1.17619) erzeugt aus einem festen Parameter einen AR-Adapter-Wert — sein unverdrahtetes INIT-Event feuert automatisch einmalig beim Kaltstart (dasselbe Prinzip wie bei den unverdrahteten INIT-Events von Q_NumericValue_PHYSA in dieser und anderen Übungen: ein nicht verbundenes Event dieser Art wird von 4diac beim Ressourcen-Start automatisch ausgelöst).
- **Korrektur:** AR_MUL_2 (generischer AR-nativer Multiplikations-FB, exakt das Muster von AR_ADD_2 aus Uebung_011b1_PHYSA, nur mit MUL statt ADD) multipliziert IN1 (der durchgereichte Sollwert von I3) mit IN2 (der Konstante aus initval_AR) — rein über AR-Adapter-Verbindungen, ganz ohne eigene REQ/CNF-Verkabelung. Der Faktor 1,17619 ist der Kehrwert von 0,8502 — einem beispielhaften Lastteiler-Verhältnis, wie es bei einem Danfoss-PVEA- Proportionalventil mit vorgeschaltetem PWM-Tiefpassfilter auftreten kann: 12 kΩ Verbraucher-Eingangsimpedanz gegen 2,115 kΩ Filter-Ausgangsimpedanz.
- **Ausgabe:** AR_MUL_2.OUT geht direkt (AR-Adapter) auf Q_NumericValue_PHYSA.rPhys, das das Ergebnis auf OutputNumber_N3_N schreibt.

Warum als SUB (Composite) und nicht als eigener .fbt (Basic-FB/ST-Funktion)? In diesem Projekt gibt es keinen einzigen selbstgeschriebenen .fbt — jede eigene Logik wird als Composite aus bestehenden Bausteinen verdrahtet, auch eine simple Multiplikation mit fester Konstante (siehe MyLib::sys::F_AI_RAW_TO_PERCENT, das ebenso einen Standard-F_MUL mit festem IN2 instanziiert statt eigenen ST-Code zu schreiben — dort im REQ/CNF-Stil, hier rein über AR-Adapter, je nachdem was die Nachbarbausteine schon anbieten).

Arbeitsauftrag

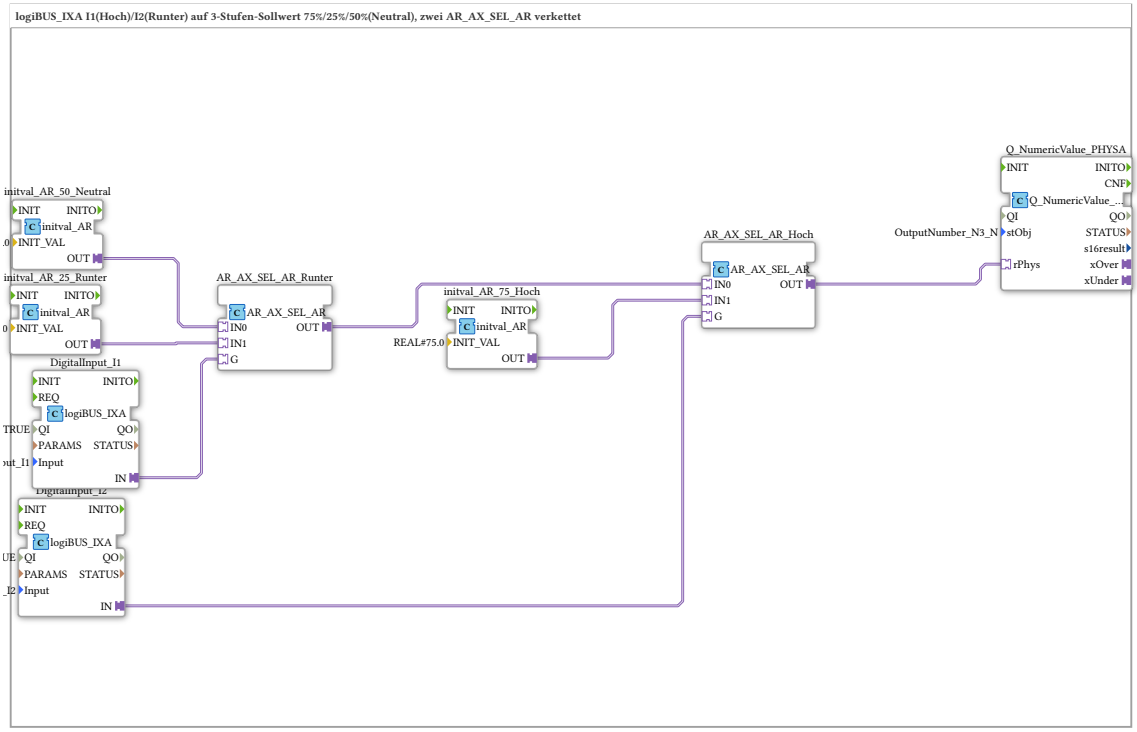
1. Legen Sie die SubApp Uebung_245_AX an (bereits als Referenzlösung vorhanden).
2. Verbinden Sie NumericValue_PHYSA.rPhys (IN1) und initval_AR.OUT (IN2) mit den beiden Sockets von AR_MUL_2, dessen OUT-Plug mit Q_NumericValue_PHYSA.rPhys.
3. Testen Sie: Geben Sie auf dem VT-Eingabefeld I3 einen Wert ein (z.B. 50,00) und prüfen Sie, dass N3 den mit dem Faktor multiplizierten Wert anzeigt (bei 1,17619: 58,81).
4. Zusatzaufgabe: Auf welche Werte korrigiert derselbe Faktor 1,17619 die Eingaben 25 % und 75 %? (Antwort: 29,40 % bzw. 88,21 % — die Korrektur ist linear und gilt mit demselben Faktor für den gesamten Bereich einheitlich, keine gesonderten INIT_VAL-Werte pro Eingabe nötig.)

Referenzlösung

Uebung_245_AX.SUB. Verwandte Übungen: Uebung_011f_PHYSA (reines Durchschleifen ohne Korrektur), Uebung_011b1_PHYSA (Verknüpfung zweier Eingabefelder über AR_ADD_2, dasselbe AR-native Verdrahtungsmuster wie hier, nur mit ADD statt MUL und ohne initval_AR).

2.1.2.543 Uebung_246_AX: logiBUS_IXA I1(Hoch)/I2(Runter) auf 3-Stufen-Sollwert 75%/25%/50%(Neutral), zwei AR_AX_SEL_AR verkettet

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_246_AX (logiBUS_IXA I1(Hoch)/I2(Runter) auf 3-Stufen-Sollwert 75%/25%/50%(Neutral), zwei AR_AX_SEL_AR verkettet).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **logiBUS_IXA I1(Hoch)/I2(Runter) auf 3-Stufen-Sollwert 75%/25%/50%(Neutral), zwei AR_AX_SEL_AR verkettet**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_246_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
- **DigitalInput_I2**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I2
- **initval_AR_50_Neutral**: Instanz des Typs adapter::types::unidirectional::AR::initval::initval_AR.
  - Parameter INIT_VAL = REAL#50.0
- **initval_AR_25_Runter**: Instanz des Typs adapter::types::unidirectional::AR::initval::initval_AR.
  - Parameter INIT_VAL = REAL#25.0
- **initval_AR_75_Hoch**: Instanz des Typs adapter::types::unidirectional::AR::initval::initval_AR.
  - Parameter INIT_VAL = REAL#75.0
- **AR_AX_SEL_AR_Runter**: Instanz des Typs adapter::iec61131::selection::AR_AX_SEL_AR.
- **AR_AX_SEL_AR_Hoch**: Instanz des Typs adapter::iec61131::selection::AR_AX_SEL_AR.
- **Q_NumericValue_PHYSA**: Instanz des Typs isobus::UT::Q::Q_NumericValue_PHYSA.
  - Parameter stObj = OutputNumber_N3_N

Verbindungen und Schnittstellen

Adapterverbindungen:

- initval_AR_50_Neutral.OUT -> AR_AX_SEL_AR_Runter.IN0
- initval_AR_25_Runter.OUT -> AR_AX_SEL_AR_Runter.IN1
- DigitalInput_I1.IN -> AR_AX_SEL_AR_Runter.G
- AR_AX_SEL_AR_Runter.OUT -> AR_AX_SEL_AR_Hoch.IN0
- initval_AR_75_Hoch.OUT -> AR_AX_SEL_AR_Hoch.IN1

- DigitalInput_I2.IN -> AR_AX_SEL_AR_Hoch.G
- AR_AX_SEL_AR_Hoch.OUT -> Q_NumericValue_PHYSA.rPhys

Hinweise aus dem Modell

Zwei verkettete Binaer-Selektoren statt eines 3-Wege-Selektors (den es fuer AR nicht gibt): innen waehlt I2 (Runter) zwischen Neutral(50) und Runter(25), aussen waehlt I1 (Hoch) zwischen diesem Ergebnis und Hoch(75). Werden I1 und I2 gleichzeitig gehalten, gewinnt I1 (Hoch), weil sein Selektor der aeussere ist - bewusste Prioritaet, siehe Beschreibung.

Aktuelle Verdrahtung: `DigitalInput_I1` (Input_I1) schaltet den inneren Selektor `AR_AX_SEL_AR_Runter` (Runter), `DigitalInput_I2` (Input_I2) den äußeren Selektor `AR_AX_SEL_AR_Hoch` (Hoch). Die Zuordnung in der Modellbeschreibung oben (I1 = Hoch, I2 = Runter) gilt daher vertauscht; bei gleichzeitigem Halten gewinnt jetzt I2 (Hoch).

Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_246_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.544 Übung 246: Hoch/Runter auf 3-Stufen-Sollwert — zwei AR_AX_SEL_AR verkettet

Thema: Digitale Taster als „3-Stufen-Joystick“, Kaskadierung von binären AR-Selektoren

Situationsbeschreibung

Viele reale Funktionen (siehe `Uebung_245_AX` und die zugehörige PVEA-Kompensationsrechnung) brauchen keinen echten stufenlosen Analogeingang, sondern nur zwei einfache Taster „Hoch“ und „Runter“, deren Zustand auf drei feste Sollwerte abgebildet wird: gedrückt „Hoch“ → oberer Wert, gedrückt „Runter“ → unterer Wert, keiner gedrückt (bzw. losgelassen) → Neutralwert in der Mitte. Das ist die digitale Vorstufe zu einem echten Proportional-Joystick — z.B. für ein Heben/Senken-Funktion mit einem Danfoss-PVEA-Proportionalventil, bei der 25 %/50 %/75 % von `U_DC` als die drei Sollwerte dienen.

Funktionsbeschreibung

- **Eingabe:** `I1` (Hoch) und `I2` (Runter) werden wie in `Uebung_001_AX` über `logiBUS_IXA` gelesen — deren `IN`-Plug ist ein AX-Adapter (kontinuierlicher Pegel, kein Tastenimpuls).
- **Feste Sollwerte als AR-Konstanten:** drei `initval_AR`-Instanzen liefern `REAL#25.0`, `REAL#50.0`, `REAL#75.0` als AR-Adapter-Werte (siehe `Uebung_245_AX` für das Prinzip).
- **Kaskadierte Auswahl:** Es gibt keinen fertigen 3-Wege-AR-Selektor, deshalb zwei `AR_AX_SEL_AR` (binäre Auswahl: `G=FALSE→IN0`, `G=TRUE→IN1`) hintereinander:
  - **Innen** (`AR_AX_SEL_AR_Runter`): `G = I2`. `IN0 = 50` (Neutral), `IN1 = 25` (Runter). Ergebnis: 50, solange `I2` losgelassen ist, sonst 25.
  - **Außen** (`AR_AX_SEL_AR_Hoch`): `G = I1`. `IN0` = Ergebnis des inneren Selektors, `IN1 = 75` (Hoch). Ergebnis: der innere Wert, solange `I1` losgelassen ist, sonst 75.
- **Ausgabe:** `Q_NumericValue_PHYSA` schreibt das Endergebnis auf `OutputNumber_N3_N`.
- **Priorität bei gleichzeitigem Drücken:** Da `I1`s Selektor der äußere ist, gewinnt „Hoch“ bei gleichzeitigem Drücken beider Taster (75 statt 25) — eine bewusste, aber willkürliche Entscheidung dieser Referenzlösung, keine zwingende Vorgabe. Genau wie bei `Uebung_229_AX` („Last-Wins“) ist der Konfliktfall hier explizit dokumentiert statt stillschweigend offen zu bleiben.

Arbeitsauftrag

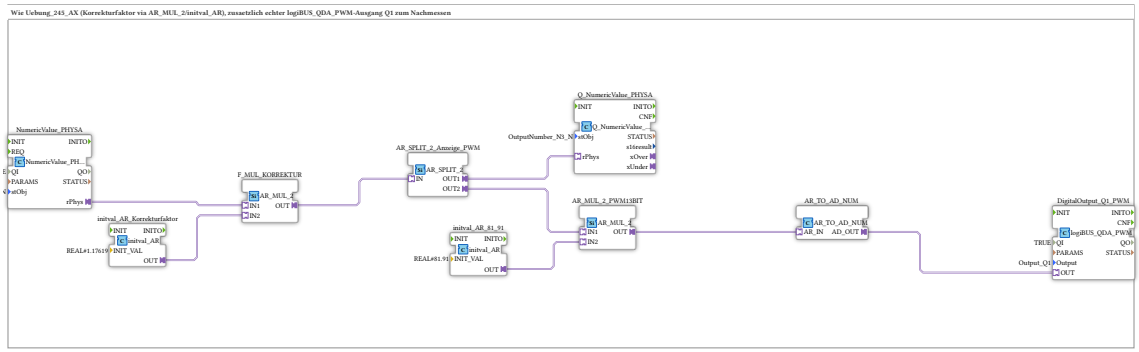
1. Legen Sie die SubApp `Uebung_246_AX` an (bereits als Referenzlösung vorhanden).
2. Verdrahten Sie die drei `initval_AR`-Konstanten und die zwei `AR_AX_SEL_AR` wie beschrieben.
3. Testen Sie: `I1` halten → Anzeige 75; loslassen → Anzeige zurück auf 50. `I2` halten → Anzeige 25; loslassen → Anzeige zurück auf 50. Beide gleichzeitig halten → Anzeige 75 (Hoch gewinnt).
4. Zusatzaufgabe: Tauschen Sie die Reihenfolge der Kaskade (Hoch innen, Runter außen) — wie ändert sich das Verhalten bei gleichzeitigem Drücken?

Referenzlösung

`Uebung_246_AX.SUB`. Verwandte Übungen: `Uebung_001_AX` (`logiBUS_IXA`-Grundmuster), `Uebung_245_AX` (`initval_AR`, feste AR-Konstanten).

2.1.2.545 Uebung_247_AX: Wie Uebung_245_AX (Korrekturfaktor via AR_MUL_2/ initval_AR), zusaetzlich echter logiBUS_QDA_PWM-Ausgang Q1 zum Nachmessen

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_247_AX (Wie Uebung_245_AX (Korrekturfaktor via AR_MUL_2/initval_AR), zusaetzlich echter logiBUS_QDA_PWM-Ausgang Q1 zum Nachmessen).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Wie Uebung_245_AX (Korrekturfaktor via AR_MUL_2/initval_AR), zusaetzlich echter logiBUS_QDA_PWM-Ausgang Q1 zum Nachmessen**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_247_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **NumericValue_PHYSA**: Instanz des Typs isobus::UT::io::NumericValue::NumericValue_PHYSA.
  - Parameter QI = TRUE
  - Parameter stObj = InputNumber_I3_N
- **F_MUL_KORREKTUR**: Instanz des Typs adapter::iec61131::arithmetic::AR_MUL_2.
- **initval_AR_Korrekturfaktor**: Instanz des Typs adapter::types::unidirectional::AR::initval::initval_AR.
  - Parameter INIT_VAL = REAL#1.17619
- **AR_SPLIT_2_Anzeige_PWM**: Instanz des Typs adapter::events::unidirectional::AR_SPLIT_2.
- **Q_NumericValue_PHYSA**: Instanz des Typs isobus::UT::Q::Q_NumericValue_PHYSA.
  - Parameter stObj = OutputNumber_N3_N
- **AR_MUL_2_PWM13BIT**: Instanz des Typs adapter::iec61131::arithmetic::AR_MUL_2.
- **initval_AR_81_91**: Instanz des Typs adapter::types::unidirectional::AR::initval::initval_AR.
  - Parameter INIT_VAL = REAL#81.91
- **DigitalOutput_Q1_PWM**: Instanz des Typs logiBUS::io::DQ::logiBUS_QDA_PWM.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1

- **AR_TO_AD_NUM**: Instanz des Typs adapter::conversion::unidirectional::AR_TO_AD_NUM.

Verbindungen und Schnittstellen

Adapterverbindungen:

- NumericValue_PHYSA.rPhys -> F_MUL_KORREKTUR.IN1
- initval_AR_Korrekturfaktor.OUT -> F_MUL_KORREKTUR.IN2
- F_MUL_KORREKTUR.OUT -> AR_SPLIT_2_Anzeige_PWM.IN
- AR_SPLIT_2_Anzeige_PWM.OUT1 -> Q_NumericValue_PHYSA.rPhys
- AR_SPLIT_2_Anzeige_PWM.OUT2 -> AR_MUL_2_PWM13BIT.IN1
- initval_AR_81_91.OUT -> AR_MUL_2_PWM13BIT.IN2
- AR_MUL_2_PWM13BIT.OUT -> AR_TO_AD_NUM.AR_IN
- AR_TO_AD_NUM.AD_OUT -> DigitalOutput_Q1_PWM.OUT

Hinweise aus dem Modell

> 0-100% Tastgrad = 0-8191 (13-Bit LEDC, siehe RampLimitFS_TO_logiBUS_QDA_PWM_OPC.SUB in MyLib_AX-1.0.0): Faktor 81,91 = 8191/100. Korrigierter Wert (bis 117,619% bei I3>85%) wird NICHT geklemmt - zum Nachmessen absichtlich so belassen, siehe Beschreibung. Mit Multimeter/Oszilloskop an Q1 gegen GND: Mittelwert = Tastgrad% x Versorgungsspannung.

Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_247_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.546 Übung 247: Korrekturfaktor mit echtem PWM-Ausgang zum Nachmessen

Thema: Von der VT-Anzeige zur echten Hardware — REAL-Prozent auf 13-Bit-PWM-Tastgrad, AR_SPLIT_2 zum Verzweigen

Situationsbeschreibung

`Uebung_245_AX` hat den Korrekturfaktor nur auf dem VT-Zahlenfeld sichtbar gemacht. Diese Übung ist eine Kopie von `Uebung_245_AX` mit einem zusätzlichen, echten `logiBUS_QDA_PWM`-Ausgang auf `Q1` — damit lässt sich der korrigierte Wert nicht nur ablesen, sondern mit Multimeter oder Oszilloskop direkt am Ausgang nachmessen. Das ist derselbe letzte Schritt, der auch in einer realen PWM-auf-PVEA-Kette (siehe `RampLimitFS_TO_logiBUS_QDA_PWM_OPC.SUB` in `MyLib_AX-1.0.0`) den Sollwert an die Hardware bringt.

Funktionsbeschreibung

- **Unverändert wie `Uebung_245_AX`:** `NumericValue_PHYSA` (`I3`) ⟶ `AR_MUL_2` (`F_MUL_KORREKTUR`, `IN2` fest über `initval_AR` auf `REAL#1,17619`) liefert den korrigierten Prozentwert.
- **Verzweigen mit `AR_SPLIT_2`:** Ein AR-Adapter-Plug kann nur an EINEN Socket angeschlossen werden (dieselbe 1:1-Regel wie bei den anderen Adapter-Typen, siehe z.B. `AX_SPLIT_n` in der `Softkey_Aux_IXA_TO_Remote_WRITE`-Familie). `AR_SPLIT_2` fächert den korrigierten Wert auf zwei Ziele auf: `OUT1` weiter zum VT-Feld `N3` (wie bisher), `OUT2` neu zur PWM-Kette.
- **Prozent auf 13-Bit-Tastgrad:** ein zweiter `AR_MUL_2` (mit `initval_AR = REAL#81.91`, das ist 8191/100) skaliert den 0-100%-Wert auf den 13-Bit-LEDC-Tastgrad 0-8191 — exakt dieselbe Umrechnung, die in `RampLimitFS_TO_logiBUS_QDA_PWM_OPC.SUB` (`MyLib_AX-1.0.0`) über `F_MUL_TO_PWM13BIT`/`F_DIV_TO_PWM13BIT` läuft, hier aber als reine AR-Multiplikation mit einem einzigen festen Faktor (da wir schon in Prozent, nicht im ISO-11783-Rohwert 0-64255, rechnen).
- **REAL nach DWORD, aber richtig herum:** `AR_TO_AD_NUM` — numerische Wandlung `REAL⟶DWORD` über einen internen `UDINT`-Zwischenschritt (`F_REAL_TO_UDINT` echter Cast, `F_UDINT_TO_DWORD` Bit-

Reinterpretation, hier unproblematisch, weil UDINT/DWORD beide 32 Bit breit sind und denselben Wertebereich abbilden). **Vorsicht:** Es gibt auch ein direktes AR_TO_AD in derselben Bibliothek — das macht aber eine reine IEEE754-Bit-Reinterpretation (für Serialisierung gedacht), keine Zahlenumwandlung, und wäre hier falsch (siehe dessen eigener Comment-Text und, spiegelbildlich für die andere Richtung, AD_TO_AR_NUMs Dokumentation). AR_TO_AD_NUM ist neu (diese Übung war der erste Anwendungsfall) und packt genau die Kette ein, die man sonst von Hand über AR_TO_AUDI + AUDI_TO_AD verdrahten müsste.

- **Echter Ausgang:** logiBUS_QDA_PWM auf Output_Q1 gibt den Tastgrad als echtes PWM-Signal aus (13-Bit-LEDC, 400 Hz laut ESP32-Firmware).

Messen: Mittelwert an Q1 gegen GND (Multimeter im Gleichspannungs-Mittelwert-Modus, oder Oszilloskop-Mittelwertfunktion) = Tastgrad% × Versorgungsspannung. Bei I3 = 50 sollte der korrigierte Tastgrad 58,81 % sein — bei z.B. 24 V Versorgung also ca. 14,1 V Mittelwert.

Achtung: Der korrigierte Wert wird hier NICHT geklemmt. Werte über I3 ≈ 85 ergeben einen Tastgrad über 100 % (>8191) — was am Ausgang passiert (Sättigung bei 100 %, Wraparound, oder ein Fehlerstatus), ist absichtlich nicht vorgegeben: das selbst zu beobachten ist Teil der Übung.

Arbeitsauftrag

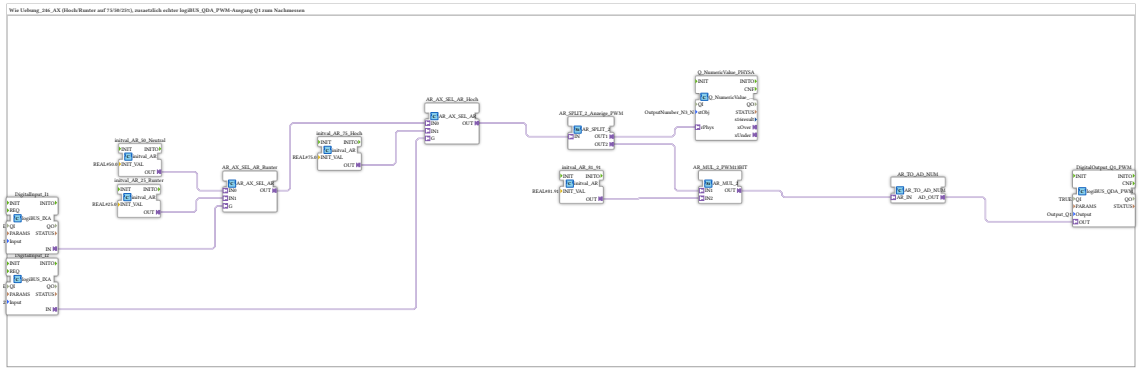
1. Legen Sie die SubApp Uebung_247_AX an (bereits als Referenzlösung vorhanden).
2. Schließen Sie ein Messgerät an Q1 gegen GND an.
3. Geben Sie auf I3 verschiedene Werte im Bereich 0-85 ein und vergleichen Sie den gemessenen Mittelwert mit der Anzeige auf N3 (korrigierter Prozentwert) unter Berücksichtigung Ihrer Versorgungsspannung.
4. Geben Sie testweise einen Wert über 85 ein und beobachten Sie, was am Ausgang passiert.

Referenzlösung

Uebung_247_AX.SUB. Basiert auf Uebung_245_AX. Verwandte Übung: Uebung_248_AX (derselbe PWM-Messaufbau, aber mit dem Hoch/Runter-Sollwert aus Uebung_246_AX).

2.1.2.547 Uebung_248_AX: Wie Uebung_246_AX (Hoch/Runter auf 75/50/25%), zusaetzlich echter logiBUS_QDA_PWM-Ausgang Q1 zum Nachmessen

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_248_AX (Wie Uebung_246_AX (Hoch/Runter auf 75/50/25%), zusaetzlich echter logiBUS_QDA_PWM-Ausgang Q1 zum Nachmessen).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Wie Uebung_246_AX (Hoch/Runter auf 75/50/25%), zusaetzlich echter logiBUS_QDA_PWM-Ausgang Q1 zum Nachmessen**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_248_AX.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1

- **DigitalInput_I2**: Instanz des Typs logiBUS::io::DI::logiBUS_IXA.
  - Parameter QI = TRUE
  - Parameter Input = Input_I2
- **initval_AR_50_Neutral**: Instanz des Typs adapter::types::unidirectional::AR::initval::initval_AR.
  - Parameter INIT_VAL = REAL#50.0
- **initval_AR_25_Runter**: Instanz des Typs adapter::types::unidirectional::AR::initval::initval_AR.
  - Parameter INIT_VAL = REAL#25.0
- **initval_AR_75_Hoch**: Instanz des Typs adapter::types::unidirectional::AR::initval::initval_AR.
  - Parameter INIT_VAL = REAL#75.0
- **AR_AX_SEL_AR_Runter**: Instanz des Typs adapter::iec61131::selection::AR_AX_SEL_AR.
- **AR_AX_SEL_AR_Hoch**: Instanz des Typs adapter::iec61131::selection::AR_AX_SEL_AR.
- **AR_SPLIT_2_Anzeige_PWM**: Instanz des Typs adapter::events::unidirectional::AR_SPLIT_2.
- **Q_NumericValue_PHYSA**: Instanz des Typs isobus::UT::Q::Q_NumericValue_PHYSA.
  - Parameter stObj = OutputNumber_N3_N
- **AR_MUL_2_PWM13BIT**: Instanz des Typs adapter::iec61131::arithmetic::AR_MUL_2.
- **initval_AR_81_91**: Instanz des Typs adapter::types::unidirectional::AR::initval::initval_AR.
  - Parameter INIT_VAL = REAL#81.91
- **DigitalOutput_Q1_PWM**: Instanz des Typs logiBUS::io::DQ::logiBUS_QDA_PWM.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
- **AR_TO_AD_NUM**: Instanz des Typs adapter::conversion::unidirectional::AR_TO_AD_NUM.

Verbindungen und Schnittstellen

Adapterverbindungen:

- initval_AR_50_Neutral.OUT -> AR_AX_SEL_AR_Runter.IN0
- initval_AR_25_Runter.OUT -> AR_AX_SEL_AR_Runter.IN1
- DigitalInput_I1.IN -> AR_AX_SEL_AR_Runter.G
- AR_AX_SEL_AR_Runter.OUT -> AR_AX_SEL_AR_Hoch.IN0
- initval_AR_75_Hoch.OUT -> AR_AX_SEL_AR_Hoch.IN1
- DigitalInput_I2.IN -> AR_AX_SEL_AR_Hoch.G
- AR_AX_SEL_AR_Hoch.OUT -> AR_SPLIT_2_Anzeige_PWM.IN
- AR_SPLIT_2_Anzeige_PWM.OUT1 -> Q_NumericValue_PHYSA.rPhys
- AR_SPLIT_2_Anzeige_PWM.OUT2 -> AR_MUL_2_PWM13BIT.IN1
- initval_AR_81_91.OUT -> AR_MUL_2_PWM13BIT.IN2
- AR_MUL_2_PWM13BIT.OUT -> AR_TO_AD_NUM.AR_IN
- AR_TO_AD_NUM.AD_OUT -> DigitalOutput_Q1_PWM.OUT

Hinweise aus dem Modell

0-100% Tastgrad = 0-8191 (13-Bit LEDC, siehe RampLimitFS_TO_logiBUS_QDA_PWM_OPC.SUB in MyLib_AX-1.0.0): Faktor 81,91 = 8191/100. Mit Multimeter/Oszilloskop an Q1 gegen GND: Mittelwert = Tastgrad% x Versorgungsspannung - bei 25/50/75% Tastgrad direkt vergleichbar mit den PVEA-Sollspannungen 0,25/0,50/0,75 x U_DC.

Aktuelle Verdrahtung: `DigitalInput_I1` schaltet `AR_AX_SEL_AR_Runter`, `DigitalInput_I2` schaltet `AR_AX_SEL_AR_Hoch` (Hoch hat als äußerer Selektor Vorrang).

Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_248_AX demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.1.2.548 Übung 248: Hoch/Runter-Sollwert mit echtem PWM-Ausgang zum Nachmessen

Thema: Von der VT-Anzeige zur echten Hardware — 3-Stufen-Sollwert auf 13-Bit-PWM-Tastgrad

Situationsbeschreibung

`Uebung_246_AX` hat die drei Sollwerte (75/50/25 %) nur auf dem VT-Zahlenfeld sichtbar gemacht. Diese Übung ist eine Kopie von `Uebung_246_AX` mit einem zusätzlichen, echten `logiBUS_QDA_PWM`-Ausgang auf `Q1` — damit lässt sich das Ergebnis direkt mit Multimeter oder Oszilloskop nachmessen, statt es nur auf dem VT zu prüfen. Zusammen mit `Uebung_247_AX` deckt das beide Bausteine einer typischen Heben/Senken-Funktion mit Taster-Sollwertvorgabe ab: dort Taster → 75/50/25 %-Sollwert (wie hier), hier zusätzlich die Umsetzung in ein echtes PWM-Signal.

Funktionsbeschreibung

- **Unverändert wie `Uebung_246_AX`:** `I1`(Hoch)/`I2`(Runter) über `logiBUS_IXA`, zwei verkettete `AR_AX_SEL_AR` liefern 75/50/25 % je nach Tasterzustand.
- **Verzweigen mit `AR_SPLIT_2`:** genau wie in `Uebung_247_AX` fächert `AR_SPLIT_2` den Sollwert auf das VT-Feld `N3` (`OUT1`, unverändert) und die neue PWM-Kette (`OUT2`) auf.
- **Prozent auf 13-Bit-Tastgrad:** `AR_MUL_2` mit `initval_AR = REAL#81.91` (= 8191/100) skaliert 0-100 % auf 0-8191 — siehe `Uebung_247_AX` für die Herleitung.
- **REAL nach DWORD:** `AR_TO_AD_NUM` (numerische Wandlung über internen `UDINT`-Schritt, siehe `Uebung_247_AX_Beschreibung.md`). Nicht das bit-reinterpretierende `AR_TO_AD` verwenden.
- **Echter Ausgang:** `logiBUS_QDA_PWM` auf `Output_Q1`.

Messen: Mittelwert an `Q1` gegen GND = Tastgrad% × Versorgungsspannung. Da hier keine Korrektur stattfindet, sollten `I1` gehalten 75 % (0,75 × U_DC), `I2` gehalten 25 % (0,25 × U_DC) und Neutralstellung 50 % (0,50 × U_DC) ergeben — direkt vergleichbar mit den erwarteten PVEA-Sollspannungen aus einer entsprechenden Kompensationsrechnung.

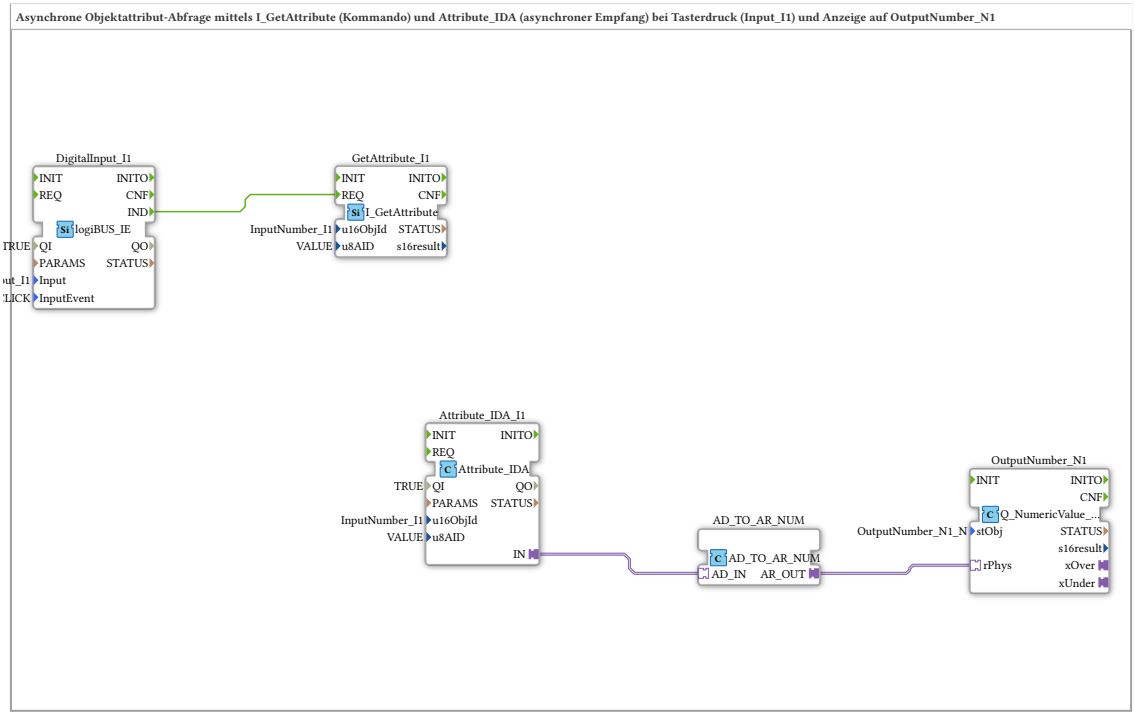
Arbeitsauftrag

1. Legen Sie die SubApp `Uebung_248_AX` an (bereits als Referenzlösung vorhanden).
2. Schließen Sie ein Messgerät an `Q1` gegen GND an.
3. Halten Sie `I1` bzw. `I2` und prüfen Sie den gemessenen Mittelwert gegen die erwarteten 75 % bzw. 25 % Ihrer Versorgungsspannung; loslassen → 50 %.
4. Vergleichen Sie mit `Uebung_247_AX`: dort wird derselbe PWM-Aufbau von einem korrigierten statt einem rohen Sollwert gespeist — messen Sie den Unterschied direkt am Ausgang.

Referenzlösung

`Uebung_248_AX.SUB`. Basiert auf `Uebung_246_AX`. Verwandte Übung: `Uebung_247_AX` (derselbe PWM-Messaufbau, aber mit dem Korrekturfaktor aus `Uebung_245_AX`).

2.1.2.549 Uebung_249_AX: Asynchrone Objektattribut-Abfrage mit I_GetAttribute und Attribute_IDA



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung fragt per Tasterdruck (`Input_I1`) ein Attribut eines VT-Objekts asynchron ab und zeigt es auf `OutputNumber_N1` an. Das Kommando wird von `I_GetAttribute` gesendet, die Antwort kommt unabhängig davon über `Attribute_IDA` an: Beide Bausteine sind nicht per Event verbunden, sondern beziehen sich über dieselben Parameter (`u16ObjId = InputNumber_I1`, `u8AID = VALUE`) auf dasselbe Objekt und Attribut. Der empfangene Wert wird über `AD_TO_AR_NUM` in einen `REAL`-Adapterwert gewandelt und mit `Q_NumericValue_PHYSA` ausgegeben.

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_I1**: Taster mit Klick-Erkennung
  - **Typ**: `logiBUS::io::DI::logiBUS_IE`
  - **Parameter**: `QI = TRUE`, `Input = Input_I1`, `InputEvent = BUTTON_SINGLE_CLICK`
  - **Erklärung**: Meldet einen Einzelklick auf `I1` als Event `IND`.
- **GetAttribute_I1**: Abfrage-Kommando
  - **Typ**: `isobus::UT::I::I_GetAttribute`
  - **Parameter**: `u16ObjId = InputNumber_I1`, `u8AID = VALUE`
  - **Erklärung**: Fordert bei `REQ` das Attribut `VALUE` des Objekts `InputNumber_I1` an.
- **Attribute_IDA_I1**: asynchroner Empfang des Attributs
  - **Typ**: `isobus::UT::io::Attribute::Attribute_IDA`
  - **Parameter**: `QI = TRUE`, `u16ObjId = InputNumber_I1`, `u8AID = VALUE`
  - **Erklärung**: Liefert den empfangenen Attributwert über den Adapter `IN`.
- **AD_TO_AR_NUM**: Konvertierung `AD` nach `AR`
  - **Typ**: `adapter::conversion::unidirectional::AD_TO_AR_NUM`
  - **Parameter**: keine
  - **Erklärung**: Wandelt den Attributwert (`AD_IN`) numerisch in einen `REAL`-Adapterwert (`AR_OUT`).
- **OutputNumber_N1**: numerische Anzeige
  - **Typ**: `isobus::UT::Q::Q_NumericValue_PHYSA`
  - **Parameter**: `stObj = OutputNumber_N1_N`
  - **Erklärung**: Zeigt den Wert aus `rPhys` am Objekt `OutputNumber_N1_N` an.

Programmablauf und Verbindungen

1. `DigitalInput_I1.IND` ⟶ `GetAttribute_I1.REQ` (EventConnection): Ein Klick auf `I1` sendet die Attribut-Abfrage.
2. `Attribute_IDA_I1.IN` ⟶ `AD_TO_AR_NUM.AD_IN` (AdapterConnection): Der asynchron empfangene Attributwert wird gewandelt.
3. `AD_TO_AR_NUM.AR_OUT` ⟶ `OutputNumber_N1.rPhys` (AdapterConnection): Der Wert wird an `OutputNumber_N1` angezeigt.

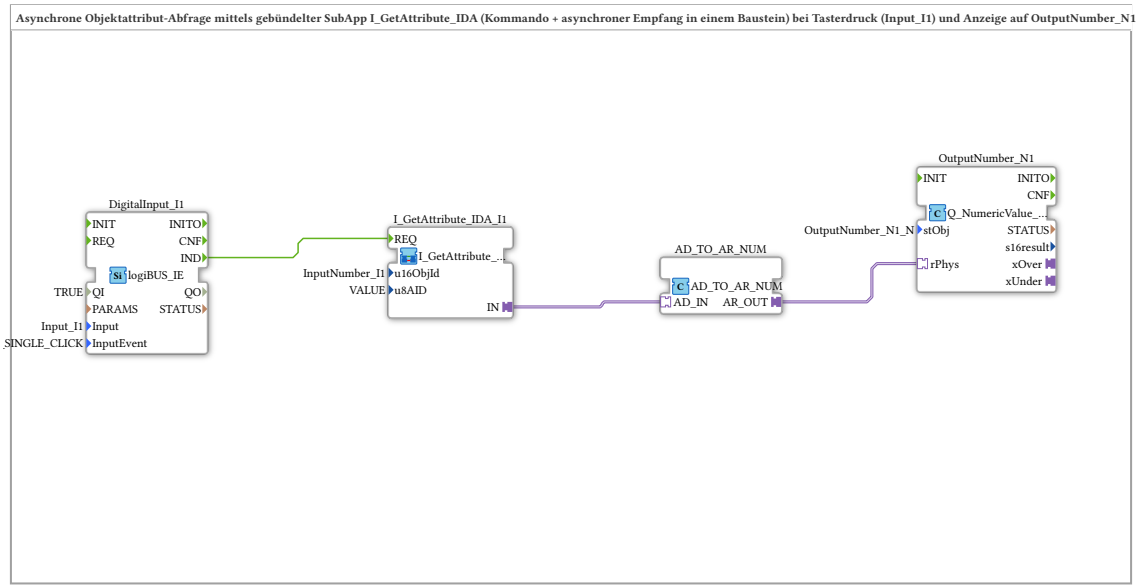
Zusammenfassung

Die Übung 249_AX zeigt, wie ein Attribut eines VT-Objekts abgefragt wird: `I_GetAttribute` sendet das Kommando bei Tasterdruck, `Attribute_IDA` empfängt die Antwort asynchron, und `AD_TO_AR_NUM` bringt den Wert für die Anzeige über `Q_NumericValue_PHYSA` in ein `REAL`-Format.

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

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

2.1.2.550 Uebung_249b_AX: Asynchrone Objektattribut-Abfrage mit gebündelter SubApp I_GetAttribute_IDA



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung fragt bei einem Tasterdruck (`Input_I1`) ein Objektattribut eines VT-Objekts asynchron ab und zeigt das Ergebnis auf `OutputNumber_N1` an. Kommando (`GetAttribute`) und asynchroner Empfang der Antwort (`Attribute_IDA`) sind dabei in der gebündelten SubApp `MyLib::sys::I_GetAttribute_IDA` zusammengefasst; sie teilen sich die Parameter `u16ObjId` und `u8AID`.

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_I1**: logiBUS Eingang mit Klick-Erkennung
  - **Typ**: `logiBUS::io::DI::logiBUS_IE`
  - **Parameter**: `QI = TRUE`, `Input = Input_I1`, `InputEvent = BUTTON_SINGLE_CLICK`
  - **Erklärung**: Meldet einen Einzelklick auf `I1` als Event `IND`.
- **I_GetAttribute_IDA_I1**: gebündelte Attributabfrage (SubApp-Instanz)
  - **Typ**: `MyLib::sys::I_GetAttribute_IDA`
  - **Parameter**: `u16ObjId = InputNumber_I1`, `u8AID = VALUE`
  - **Erklärung**: Bei `REQ` sendet der interne Baustein `I_GetAttribute` das Kommando GetAttribute (F.58) für das Attribut `u8AID` des Objekts `u16ObjId`; der interne Baustein `Attribute_IDA` empfängt die

Antwort asynchron und stellt sie als `AD`-Adapter (DWORD) am Ausgang `IN` bereit. Hier wird der Wert (`VALUE`) des Eingabefelds `InputNumber_I1` abgefragt.

- **AD_TO_AR_NUM**: Adapterkonverter DWORD → REAL
  - **Typ**: `adapter::conversion::unidirectional::AD_TO_AR_NUM`
  - **Parameter**: keine
  - **Erklärung**: Wandelt den `AD`-Adapter numerisch (über UDINT, keine Bit-Umdeutung) in einen `AR`-Adapter (REAL) um.
- **OutputNumber_N1**: Ausgabe auf das VT-Zahlenfeld
  - **Typ**: `isobus::UT::Q::Q_NumericValue_PHYSA`
  - **Parameter**: `stObj = OutputNumber_N1_N`
  - **Erklärung**: Schreibt den physikalischen REAL-Wert (`rPhys`) in das VT-Objekt aus `stObj` (Ausgabefeld `OutputNumber_N1`).

Programmablauf und Verbindungen

1. `DigitalInput_I1.IND` → `I_GetAttribute_IDA_I1.REQ`: der Tasterklick löst die Attributabfrage aus.
2. `I_GetAttribute_IDA_I1.IN` → `AD_TO_AR_NUM.AD_IN`: die asynchron eintreffende Antwort wird als DWORD-Adapter weitergegeben und numerisch in REAL gewandelt.
3. `AD_TO_AR_NUM.AR_OUT` → `OutputNumber_N1.rPhys`: der Wert wird auf `OutputNumber_N1` angezeigt.

Verhalten: Jeder Klick auf `I1` fragt den aktuellen Wert des Eingabefelds `InputNumber_I1` vom Terminal ab; sobald die Antwort eintrifft, erscheint er auf `OutputNumber_N1`.

Zusammenfassung

Die Übung zeigt, wie sich Kommando und asynchroner Empfang einer Attributabfrage in einer einzigen SubApp bündeln lassen, sodass im Netzwerk nur noch ein Baustein mit gemeinsamer Objekt- und Attribut-ID nötig ist, und wie das Ergebnis über einen Adapterkonverter auf einem VT-Zahlenfeld ausgegeben wird.

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

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

2.1.2.551 ISOBUS, Softkeys und Terminals

Diese Übungen behandeln die Integration der Steuerung in das ISOBUS-Netzwerk, einschließlich der Bedienung über virtuelle Tasten (Softkeys/Buttons) und AUX-Eingaben (Joystick).

🎧 **Podcast**

- ISO 11783-6: Softkeys und das Virtual Terminal verstehen – Dein Schlüssel zur Landmaschinen-Mechatronik

2.1.2.552 Basis-Logik und direkte Verbindungen

Diese Übungen behandeln die grundlegende Anbindung von I/Os und einfache logische Verknüpfungen (AND, OR, XOR) unter Verwendung des AX-Adapters.

2.1.2.553 Speicherelemente und Klick-Events

Hier lernen Sie den Umgang mit verschiedenen Flip-Flop-Typen (SR, RS, Toggle) und die Auswertung komplexer Taster-Ereignisse wie Doppel- oder Langklicks.

2.1.2.554 🛠️ Übungen (Arbeitsbereich test_AX)

Hier finden Sie die Dokumentation der Übungen für den logiBUS® mit dem **AX Adapter**. Die Übungen sind thematisch gruppiert, um den Lernfortschritt optimal zu unterstützen.

🎧 **Podcast**

- DIN EN 61499-1 Entschlüsselt: Der Bauplan für modulare, verteilte Steuerungssysteme
- FNIRSI DWS-200 Lötstation im Test: 200 Watt Power und die Wahrheit aus den Nutzer-Reviews
- 4diac IDE: Dein Open-Source-Werkzeugkasten für verteilte Industrieautomatisierung nach IEC 61499
- 4diac-Präsentation: Zielgruppen, Struktur und Alleinstellungsmerkmal Schärfen

- Den Software-Drachen zähmen: Industrielle Automatisierung und die Zukunft der Produktion

📺 Video

- Der denkende Traktor
- Der Traktor-Vordenker
- Die Kunst des Lötens
- EAGLE CAD: A Legend
- Ideale Dioden

Vollständiger Index

Falls Sie eine bestimmte Übung anhand ihrer Nummer suchen, nutzen Sie bitte die numerische Übersicht:

- Numerischer Index aller Übungen (test_AX)

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

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

2.1.2.555 Sequenzer und Schrittketten

Diese Übungen demonstrieren die Implementierung von Ablaufsteuerungen (Schrittketten) für sequentielle Prozesse.

2.1.2.556 Signalverarbeitung, PWM und Regelung

Hier finden Sie Übungen zu komplexeren Themen wie Multiplexing, Pulsweitenmodulation (PWM), Impulszählung und mathematischen Berechnungen für Regelkreise.

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

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

2.1.2.557 Wiederverwendbare Sub-Applikationen

In diesem Abschnitt werden die Definitionen und die Funktionsweise der in den Übungen verwendeten Baustein-Typen (Typed SubApps) detailliert beschrieben.

2.1.2.558 Timer und Zeitfunktionen

In diesem Abschnitt werden Zeitverzögerungen (Einschalt-/Ausschaltverzögerung) und Taktgeber (Blinker) behandelt.

2.1.2.559 Numerischer Index (test_AX)

Hier finden Sie alle Übungen des Arbeitsbereichs `test_AX` in aufsteigender Reihenfolge.

2.2 test B

2.2.1 Numerischer Index (test_B)

Hier finden Sie alle Übungen des Arbeitsbereichs `test_B` in aufsteigender Reihenfolge.

- Uebung_000
- Uebung_000b
- Uebung_001
- Uebung_001c
- Uebung_001c2
- Uebung_001c3
- Uebung_001c4
- Uebung_001d

- Uebung_001d2
- Uebung_001e
- Uebung_001f
- Uebung_001g
- Uebung_002
- Uebung_002a
- Uebung_002a2
- Uebung_002a3
- Uebung_002a4
- Uebung_002a4b
- Uebung_002a5b
- Uebung_002b2
- Uebung_002b3
- Uebung_003
- Uebung_003a
- Uebung_003a0
- Uebung_003a_sub
- Uebung_003b
- Uebung_003b2
- Uebung_003b2_sub
- Uebung_003b3
- Uebung_003b_sub
- Uebung_003c
- Uebung_003c_sub
- Uebung_003d
- Uebung_003e
- Uebung_004a
- Uebung_004a2
- Uebung_004a2_2
- Uebung_004a2_3
- Uebung_004a2_3b
- Uebung_004a3
- Uebung_004a4
- Uebung_004a5
- Uebung_004a6
- Uebung_004a6a
- Uebung_004a7
- Uebung_004a8
- Uebung_004a9
- Uebung_004a10a
- Uebung_004a10b
- Uebung_004a11a
- Uebung_004a11b
- Uebung_004b
- Uebung_004b2
- Uebung_004b2b
- Uebung_004b2b_sub
- Uebung_004b3
- Uebung_004b3b
- Uebung_004b3b_sub
- Uebung_004b3c
- Uebung_004b3c_sub
- Uebung_004b4c
- Uebung_004b4c_sub
- Uebung_004b4d
- Uebung_004b5
- Uebung_004b6
- Uebung_004c1
- Uebung_004c2

- Uebung_004c3
- Uebung_004c4
- Uebung_004c5
- Uebung_004c6
- Uebung_004c7
- Uebung_004d_T
- Uebung_005
- Uebung_006
- Uebung_006a
- Uebung_006a2
- Uebung_006a2b
- Uebung_006a3
- Uebung_006a3_sub
- Uebung_006a4
- Uebung_006a5
- Uebung_006b
- Uebung_006c
- Uebung_006d
- Uebung_006e1
- Uebung_006e2
- Uebung_006f_RS
- Uebung_006f_SR
- Uebung_007
- Uebung_007a1
- Uebung_007a2
- Uebung_007a3
- Uebung_007b
- Uebung_007c
- Uebung_007d
- Uebung_007d2
- Uebung_007d3
- Uebung_007d4
- Uebung_008
- Uebung_009
- Uebung_009a
- Uebung_010
- Uebung_010a
- Uebung_010a2
- Uebung_010a3
- Uebung_010a4
- Uebung_010a5
- Uebung_010b1
- Uebung_010b2
- Uebung_010b3
- Uebung_010b4
- Uebung_010b4_sub
- Uebung_010b5
- Uebung_010b5_sub
- Uebung_010b6
- Uebung_010b7
- Uebung_010b8
- Uebung_010b9
- Uebung_010bA
- Uebung_010bA2
- Uebung_010bA3
- Uebung_010bA4
- Uebung_010c
- Uebung_010c2
- Uebung_010c3

- Uebung_010c3_sub
- Uebung_010c4
- Uebung_010c4_sub
- Uebung_010d
- Uebung_010d2
- Uebung_010e
- Uebung_010e2
- Uebung_010f
- Uebung_010f2
- Uebung_010f3
- Uebung_010f4
- Uebung_011
- Uebung_011a
- Uebung_011a2
- Uebung_011b1
- Uebung_011b1_PHYS
- Uebung_011b2
- Uebung_011b3
- Uebung_011b3_PHYS
- Uebung_011b4
- Uebung_011b5
- Uebung_011b6
- Uebung_011b7
- Uebung_011c
- Uebung_011d
- Uebung_011e_MIX
- Uebung_011f
- Uebung_011g1
- Uebung_012
- Uebung_012a
- Uebung_012a_sub
- Uebung_012b
- Uebung_012c
- Uebung_012c_sub
- Uebung_012d
- Uebung_012e
- Uebung_012e_sub
- Uebung_012f
- Uebung_012f_sub
- Uebung_012g
- Uebung_012j
- Uebung_012k
- Uebung_012k_sub
- Uebung_012l
- Uebung_012m
- Uebung_012m_sub
- Uebung_12x_sub
- Uebung_013
- Uebung_014
- Uebung_015
- Uebung_015a
- Uebung_015b
- Uebung_016
- Uebung_016a
- Uebung_016b
- Uebung_016c
- Uebung_017
- Uebung_018
- Uebung_018a

- Uebung_019
- Uebung_019a
- Uebung_019b
- Uebung_019c
- Uebung_020a
- Uebung_020b
- Uebung_020c
- Uebung_020c2
- Uebung_020c2_sub
- Uebung_020c3
- Uebung_020c4
- Uebung_020d
- Uebung_020e
- Uebung_020e2
- Uebung_020f
- Uebung_020f2
- Uebung_020f3
- Uebung_020f4
- Uebung_020g
- Uebung_020h
- Uebung_020i
- Uebung_021
- Uebung_022
- Uebung_023
- Uebung_024
- Uebung_025
- Uebung_026
- Uebung_026_sub
- Uebung_028
- Uebung_028a
- Uebung_029
- Uebung_030
- Uebung_031
- Uebung_032
- Uebung_033
- Uebung_033_sub
- Uebung_034
- Uebung_034a1_Q1
- Uebung_034a1_Q2
- Uebung_034a1_Q4
- Uebung_034b
- Uebung_035
- Uebung_035a
- Uebung_035a2
- Uebung_035a3
- Uebung_035b
- Uebung_035c
- Uebung_036
- Uebung_036b
- Uebung_037
- Uebung_038
- Uebung_039
- Uebung_039_sub_NumbAnzeig
- Uebung_039_sub_Outputs
- Uebung_039a
- Uebung_039a_sub_Outputs
- Uebung_039b
- Uebung_040
- Uebung_040_2

- Uebung_041
- Uebung_042
- Uebung_043
- Uebung_044
- Uebung_045
- Uebung_046
- Uebung_047
- Uebung_048
- Uebung_049
- Uebung_051
- Uebung_052
- Uebung_053
- Uebung_054
- Uebung_055
- Uebung_056
- Uebung_057
- Uebung_058
- Uebung_059
- Uebung_060
- Uebung_060_sub_Outputs
- Uebung_061
- Uebung_062
- Uebung_063
- Uebung_070
- Uebung_070b
- Uebung_070c
- Uebung_071
- Uebung_071a
- Uebung_071b
- Uebung_072
- Uebung_072b
- Uebung_072c
- Uebung_072d
- Uebung_073
- Uebung_074
- Uebung_074b
- Uebung_075
- Uebung_076
- Uebung_079
- Uebung_080
- Uebung_080b
- Uebung_080c
- Uebung_080d
- Uebung_080e
- Uebung_080e2
- Uebung_080e3
- Uebung_080e4
- Uebung_080f
- Uebung_081
- Uebung_082
- Uebung_083
- Uebung_084
- Uebung_085
- Uebung_086
- Uebung_087
- Uebung_087a1
- Uebung_087a2
- Uebung_088
- Uebung_089

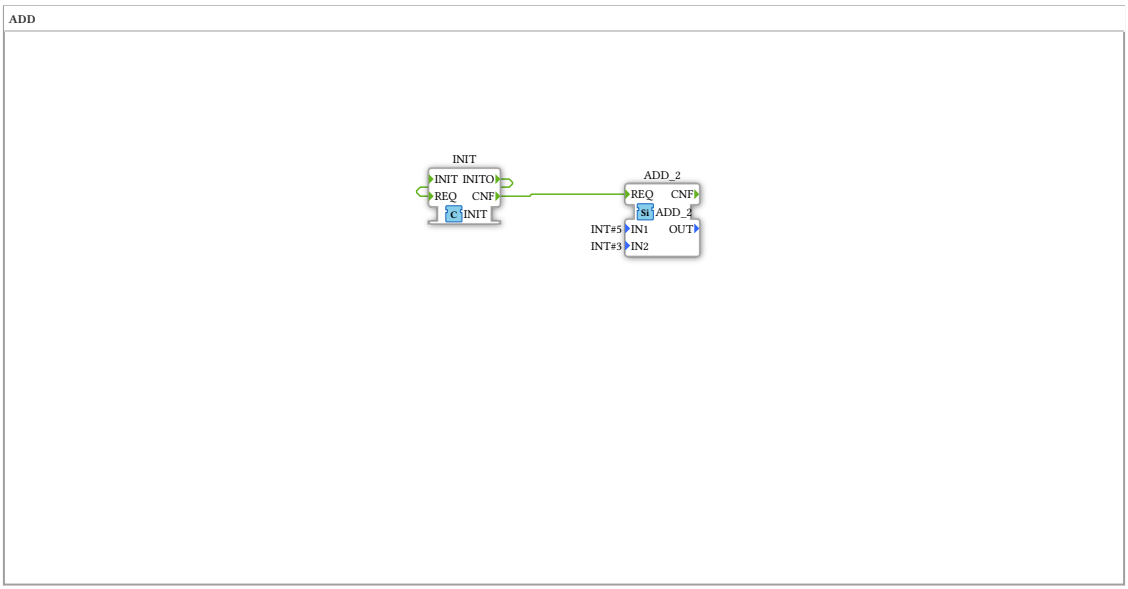
- Uebung_089a
- Uebung_090a1
- Uebung_090a2
- Uebung_091
- Uebung_093
- Uebung_093b
- Uebung_094
- Uebung_094a
- Uebung_095
- Uebung_103
- Uebung_110
- Uebung_111
- Uebung_120
- Uebung_120x_sub
- Uebung_121
- Uebung_122
- Uebung_122b
- Uebung_123
- Uebung_124
- Uebung_125
- Uebung_126
- Uebung_126b
- Uebung_126b2
- Uebung_126b2_sub
- Uebung_126b_sub
- Uebung_127
- Uebung_128
- Uebung_128b
- Uebung_130
- Uebung_131
- Uebung_132
- Uebung_133
- Uebung_134
- Uebung_135
- Uebung_140
- Uebung_150
- Uebung_151
- Uebung_152
- Uebung_153
- Uebung_160
- Uebung_160b
- Uebung_160b2
- Uebung_170
- Uebung_171
- Uebung_172
- Uebung_173
- Uebung_174
- Uebung_175
- Uebung_176
- Uebung_177
- Uebung_178
- Uebung_179
- Uebung_180
- Uebung_201
- Uebung_201_Interlock
- Uebung_201_Interlock_BOOL
- Uebung_201b
- Uebung_202
- Uebung_202b

- Uebung_203
- Uebung_203b
- Uebung_204
- Uebung_204b
- Uebung_205
- Uebung_205b
- Uebung_206
- Uebung_206b
- Uebung_207
- Uebung_208
- Uebung_209
- Uebung_210
- Uebung_210b
- Uebung_211
- Uebung_211b
- Uebung_212
- Uebung_212b
- Uebung_213
- Uebung_213b
- Uebung_213c
- Uebung_214
- Uebung_214b
- Uebung_215
- Uebung_215b
- Uebung_216
- Uebung_216b
- Uebung_217
- Uebung_217b
- Uebung_218
- Uebung_218b
- Uebung_219
- Uebung_219b
- Uebung_220
- Uebung_220b
- Uebung_221
- Uebung_221b
- Uebung_222
- Uebung_222b
- Uebung_223
- Uebung_223b
- Uebung_224
- Uebung_224b
- Uebung_225
- Uebung_225_Beschreibung
- Uebung_225b
- Uebung_225b_Beschreibung
- Uebung_226
- Uebung_226_Beschreibung
- Uebung_229
- Uebung_230
- Uebung_Isobus_Softkeys_Index
- Uebungen_Basis_Index
- Uebungen_doc_Index
- Uebungen_FlipFlops_Index
- Uebungen_Index
- Uebungen_Sequenzer_Index
- Uebungen_Signale_Index
- Uebungen_SubApps_Index
- Uebungen_Timer_Index

2.2.1.1 Uebung_000: ADD

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_000`. Dies ist das absolute Einstiegsbeispiel für die mathematische Datenverarbeitung.

Podcast



- 3000 Watt Lüge Die TVS Diode entschlüsselt
- Hannes' Turbo-Mais: Wie ein Landwirt mit Hackschnitzel-Kreislauf und Turmtrockner 15.000 Tonnen Körnermais verarbeitet

---

Ziel der Übung

Verwendung eines Standard-Mathematikbausteins (`ADD_2`). Es wird gezeigt, wie konstante Werte an die Eingänge eines Bausteins angelegt werden, um eine einfache Berechnung auszuführen.

Beschreibung und Komponenten

In `Uebung_000.SUB` wird ein Additions-Baustein zur Berechnung einer Summe genutzt.

Funktionsbausteine (FBs)

- **`ADD_2`**: Ein Baustein aus der IEC 61131-Bibliothek (Arithmetik).
- **Parameter**:
  - `IN1`: Festwert 5 (`INT#5`).
  - `IN2`: Festwert 3 (`INT#3`).

Funktionsweise

Der Baustein nimmt die beiden Eingangswerte und addiert sie intern. Da in diesem minimalistischen Beispiel keine Ereignisverbindungen definiert sind, handelt es sich um eine rein statische Berechnung des Datenflusses. Das mathematische Ergebnis am Ausgang `OUT` ist 8.

Lernziel

Diese Übung dient dazu, sich mit der 4diac-Oberfläche vertraut zu machen:

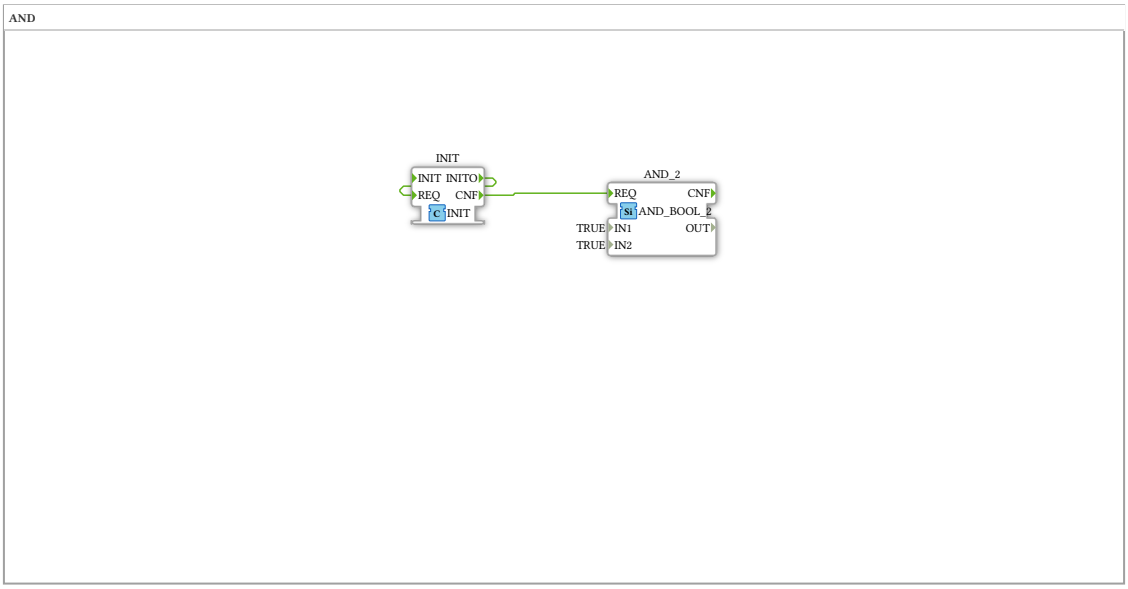
1. Bausteine aus der Bibliothek ziehen.
2. Eigenschaften (Parameter) von Bausteinen im Properties-Fenster editieren.
3. Den Unterschied zwischen variablen Eingängen und Konstanten verstehen.

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

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

2.2.1.2 Uebung_000b: AND

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_000b (AND).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **AND**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_000b.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **AND_2**: Instanz des Typs iec61131::booleanOperators::AND_BOOL_2.
  - Parameter IN1 = TRUE
  - Parameter IN2 = TRUE
- **INIT**: Instanz des Typs iec61131::booleanOperators::INIT.

Verbindungen und Schnittstellen

Ereignisverbindungen:

- INIT.INITO -> INIT.REQ
- INIT.CNF -> AND_2.REQ

Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

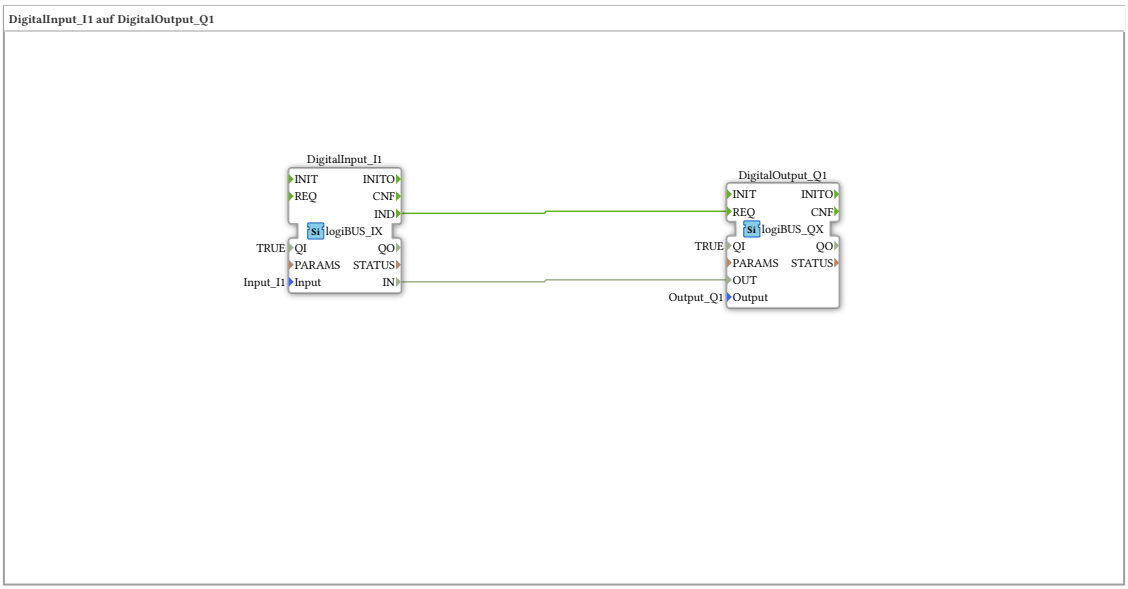
Zusammenfassung

Die Übung Uebung_000b demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.2.1.3 Uebung_001: DigitalInput_I1 auf DigitalOutput_Q1

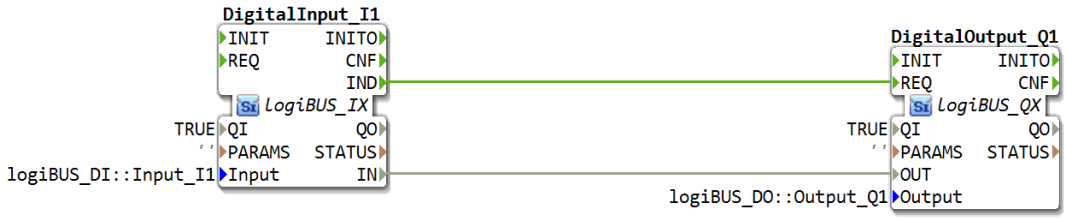
Dieser Artikel beschreibt die grundlegende logiBUS®-Übung `Uebung_001`. Hier wird das fundamentale Prinzip der IEC 61499 demonstriert: Die explizite Trennung von Datenfluss und Ereignisfluss.

Podcast



- Analyse der Novellierung der Meisterprüfungsverordnung im Land- und Baumaschinenmechatroniker-Handwerk: Ein Detaillierter Vergleich der Verordnungen von 2024 und 2001





Ziel der Übung

Das Ziel dieser Einstiegsübung ist es, ein Signal von einem physischen digitalen Eingang zu einem digitalen Ausgang zu leiten. Dabei lernen die Anwender, dass in der IEC 61499 eine reine Datenverbindung (die „Leitung“) nicht ausreicht – es muss auch ein Ereignis (der „Trigger“) übertragen werden, damit der Zielbaustein die Daten verarbeitet.

Beschreibung und Komponenten

Die Übung besteht aus einer Subapplikation (`Uebung_001.SUB`), die einen Eingangsbaustein und einen Ausgangsbaustein über zwei separate Verbindungstypen verknüpft.

Funktionsbausteine (FBs)

- **`DigitalInput_I1`**: Eine Instanz des Typs `logiBUS_IX`. Dieser Baustein repräsentiert den physischen Eingang `Input_I1`. Er stellt sowohl den logischen Zustand (`IN`) als auch ein Benachrichtigungs-Ereignis (`IND`) zur Verfügung.
- **`DigitalOutput_Q1`**: Eine Instanz des Typs `logiBUS_QX`. Dieser Baustein steuert den physischen Ausgang `Output_Q1`. Er benötigt einen Datenwert (`OUT`) und einen Auslöse-Befehl (`REQ`).

Funktionsweise

Die Logik wird durch zwei parallele Verbindungen realisiert. Der Aufbau in `Uebung_001.SUB` verdeutlicht dies:

```
<EventConnections>
    <Connection Source="DigitalInput_I1.IND" Destination="DigitalOutput_Q1.REQ"/>
</EventConnections>
<DataConnections>
    <Connection Source="DigitalInput_I1.IN" Destination="DigitalOutput_Q1.OUT"/>
</DataConnections>
```

Der Prozess läuft wie folgt ab:

1. Der Baustein `DigitalInput_I1` erkennt eine Änderung am Hardware-Eingang `I1`.
2. Er aktualisiert seinen Daten-Ausgang `IN` mit dem neuen Wert (TRUE oder FALSE).
3. Zeitgleich feuert er ein Ereignis am Port `IND` (Indication) ab.
4. Dieses Ereignis wandert über die **Event Connection** zum Port `REQ` (Request) des Bausteins `DigitalOutput_Q1`.
5. Erst durch den Empfang des Ereignisses liest `DigitalOutput_Q1` den Wert, der an seinem Port `OUT` über die **Data Connection** anliegt, und schaltet den Hardware-Ausgang entsprechend.

Anwendungsbeispiel

Ein **Lichtschalter im Haus**: Der Schalter an der Wand ist der Eingang `I1`, die Glühbirne an der Decke ist der Ausgang `Q1`. Das Kabel überträgt den Strom (Daten), aber erst das Umlegen des Schalters (Ereignis) sorgt dafür, dass die Information „An“ oder „Aus“ verarbeitet und umgesetzt wird.

2.2.1.4 Uebung_001c: DigitalInput_I1 auf DigitalOutput_Q1 –> Eingang abfragen bei Boot



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_001c`. Hier wird demonstriert, wie ein digitaler Eingang unmittelbar nach dem Systemstart (Boot-Vorgang) abgefragt wird, um den initialen Zustand an einen digitalen Ausgang zu übertragen, unter Verwendung von Standard-Ereignis- und Datenverbindungen.

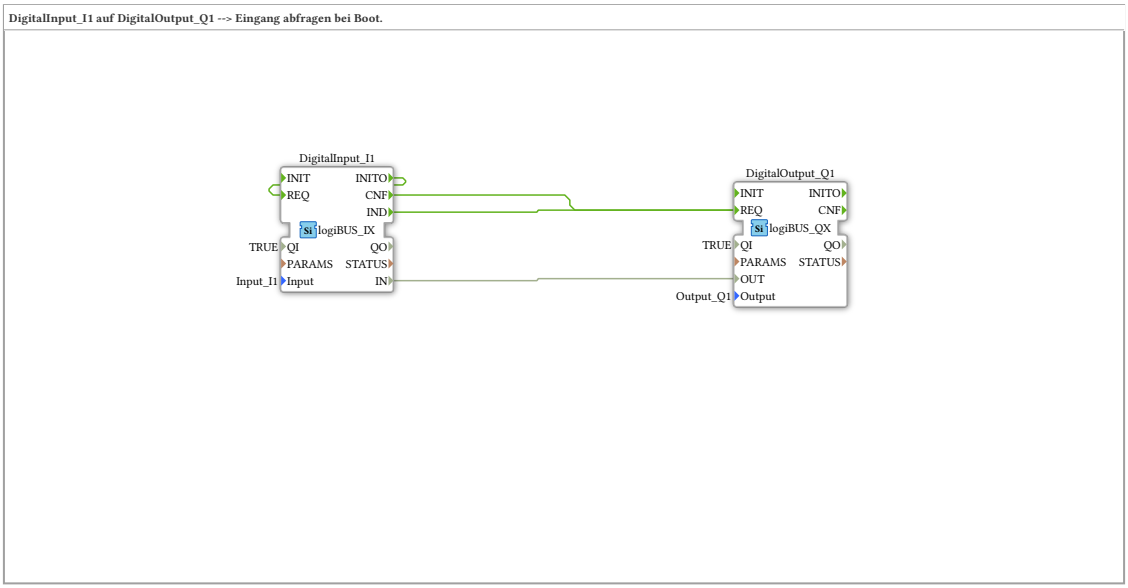
Ziel der Übung

Das Hauptziel dieser Übung ist das Verständnis des Initialisierungsvorgangs in der IEC 61499. Es soll sichergestellt werden, dass der Ausgang bereits beim Hochfahren der Steuerung den korrekten Ist-Zustand des Hardware-Eingangs übernimmt, auch wenn zu diesem Zeitpunkt noch keine Zustandsänderung (Flanke) stattgefunden hat.

Beschreibung und Komponenten

Die Übung nutzt die Subapplikation `Uebung_001c.SUB`, um eine Verbindung zwischen einem digitalen Eingang und einem Ausgang herzustellen, ergänzt um eine Selbst-Triggerung für den Systemstart.

Funktionsbausteine (FBs)



- **`DigitalInput_I1`**: Eine Instanz des Typs `logiBUS_IX`. Dieser Baustein liefert das Ereignis `IND` bei Änderungen und reagiert auf den Befehl `REQ`, um den aktuellen Wert manuell auszulesen.
- **`DigitalOutput_Q1`**: Eine Instanz des Typs `logiBUS_QX`. Dieser Baustein setzt den Hardware-Ausgang `Output_Q1` bei jedem eintreffenden `REQ`-Ereignis.

Funktionsweise

Die Logik kombiniert die normale Signalweiterleitung mit einer Initialisierungsschleife. Der Aufbau in `Uebung_001c.SUB` ist wie folgt definiert:

```
<EventConnections>
    <Connection Source="DigitalInput_I1.IND" Destination="DigitalOutput_Q1.REQ"/>
    <Connection Source="DigitalInput_I1.INITO" Destination="DigitalInput_I1.REQ"/>
    <Connection Source="DigitalInput_I1.CNF" Destination="DigitalOutput_Q1.REQ"/>
</EventConnections>
<DataConnections>
    <Connection Source="DigitalInput_I1.IN" Destination="DigitalOutput_Q1.OUT"/>
</DataConnections>
```

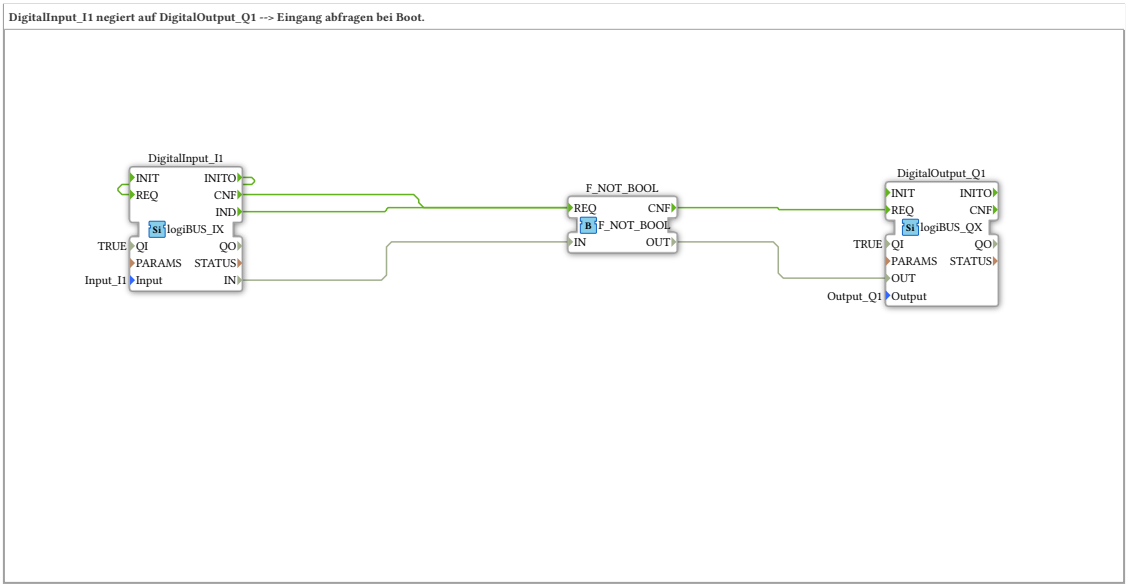
Der Ablauf gliedert sich in zwei Phasen:

1. **Initialisierungsphase (Boot)**:
   - Beim Systemstart wird der Baustein `DigitalInput_I1` initialisiert und sendet ein `INITO`-Ereignis.
   - Dieses Ereignis wird auf den eigenen `REQ`-Eingang zurückgeführt.
   - Dadurch liest der Baustein sofort den physischen Zustand ein und quittiert dies mit einem `CNF`-Ereignis.
   - Das `CNF`-Ereignis triggert schließlich `DigitalOutput_Q1.REQ`, wodurch der Ausgang bereits beim Start den korrekten Wert erhält.
2. **Betriebsphase (Laufzeit)**:
   - Jede spätere Änderung am Eingang triggert über `IND -> REQ` direkt den Ausgang, wie in Übung 001.

Anwendungsbeispiel

Ein **Zustands-Display**: Stellen Sie sich vor, der Ausgang `Q1` steuert eine Kontrollleuchte, die anzeigt, ob ein Sicherheitsschalter (`I1`) geschlossen ist. Wenn die Anlage nach einem Stromausfall neu startet, muss die Lampe sofort korrekt leuchten – nicht erst, wenn jemand den Sicherheitsschalter erneut betätigt. Die Boot-Abfrage garantiert diese sofortige Korrektheit der Anzeige.

2.2.1.5 Uebung_001c2: DigitalInput_I1 negiert auf DigitalOutput_Q1 –> Eingang abfragen bei Boot



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung realisiert eine einfache Signalverarbeitung: Der digitale Eingang `I1` wird negiert und auf den digitalen Ausgang `Q1` ausgegeben. Der Eingang wird beim Booten des Systems einmal abgefragt. Ein spezieller Kommentar im Netzwerk weist darauf hin, dass das Verhalten beim Start vom Vorhandensein einer bestimmten Ereignisverbindung (`INITO → REQ`) abhängt.

Verwendete Funktionsbausteine (FBs)

`DigitalInput_I1`

- **Typ**: `logiBUS::io::DI::logiBUS_IX`
- **Parameter**:
  - `QI = TRUE`
  - `Input = Input_I1`
- **Ereignisschnittstelle**:
  - Eingang: `REQ` (Anforderung zum Lesen des Eingangs)
  - Ausgang: `IND` (Bestätigung: Eingang gelesen), `INITO` (Initialisierungsbestätigung)
- **Datenschnittstelle**:
  - Ausgang: `IN` (aktueller digitaler Wert des Eingangs)
- **Funktionsweise**: Der Baustein liest den digitalen Wert des konfigurierten Eingangs (hier `Input_I1`) aus. Gesteuert wird das Lesen über das Ereignis `REQ`. Nach erfolgreichem Lesen wird am Ausgang `IND` ein Ereignis ausgegeben und der gelesene Wert über `IN` bereitgestellt.

`F_NOT_BOOL`

- **Typ**: `iec61131::bitwiseOperators::F_NOT_BOOL`
- **Parameter**: Keine.
- **Ereignisschnittstelle**:
  - Eingang: `REQ` (Anforderung zur Negation)
  - Ausgang: `CNF` (Bestätigung: Negation ausgeführt)
- **Datenschnittstelle**:
  - Eingang: `IN` (BOOL-Wert, der negiert werden soll)
  - Ausgang: `OUT` (negierten BOOL-Wert)
- **Funktionsweise**: Der Baustein führt eine logische Negation auf den anliegenden BOOL-Wert durch. Bei einem Ereignis an `REQ` wird der Wert an `IN` negiert und das Ergebnis an `OUT` ausgegeben. Anschließend wird `CNF` ausgelöst.

`DigitalOutput_Q1`

- **Typ**: `logiBUS::io::DQ::logiBUS_QX`
- **Parameter**:
  - `QI = TRUE`
  - `Output = Output_Q1`
- **Ereignisschnittstelle**:
  - Eingang: `REQ` (Anforderung zum Setzen des Ausgangs)
  - Ausgang: `CNF` (Bestätigung: Ausgang gesetzt)
- **Datenschnittstelle**:
  - Eingang: `OUT` (Wert, der auf den Ausgang geschrieben werden soll)
- **Funktionsweise**: Der Baustein setzt den digitalen Ausgang `Output_Q1` auf den über `OUT` erhaltenen Wert. Bei einem Ereignis an `REQ` wird der Wert übernommen und der Ausgang physisch aktualisiert. Nach Abschluss wird `CNF` ausgelöst.

Programmablauf und Verbindungen

Die drei Bausteine sind wie folgt miteinander verbunden:

1. **Eingangslesen und Negation auslösen** Der Ereignisausgang `IND` von `DigitalInput_I1` ist direkt mit dem Ereigniseingang `REQ` von `F_NOT_BOOL` verbunden. Dadurch wird nach jedem erfolgreichen Einlesen des digitalen Eingangs sofort die Negation des gelesenen Wertes angestoßen. Gleichzeitig wird der Datenwert `IN` von `DigitalInput_I1` auf den Dateneingang `IN` von `F_NOT_BOOL` übertragen.
2. **Negation und Ausgang setzen** Der Ereignisausgang `CNF` von `F_NOT_BOOL` ist mit dem Ereigniseingang `REQ` von `DigitalOutput_Q1` verbunden. Sobald die Negation abgeschlossen ist, wird der negierte Wert (vom Ausgang `OUT` von `F_NOT_BOOL`) auf den Dateneingang `OUT` von `DigitalOutput_Q1` gelegt und der Ausgang aktualisiert.
3. **Besonderheit beim Booten** Ein wichtiger Aspekt ist die Initialisierung. Der Ereignisausgang `INITO` von `DigitalInput_I1` ist mit dem Ereigniseingang `REQ` von `DigitalInput_I1` (also dem Baustein selbst) zurückverbunden. Diese Verbindung sorgt dafür, dass der Eingang unmittelbar nach dem Booten einmal gelesen wird. Ohne diese Rückkopplung würde der Ausgang `Q1` beim Start den Wert `FALSE` behalten, weil kein erstes Ereignis ausgelöst wird. **Mit der Verbindung** wird der Eingang sofort abgefragt, negiert und der Ausgang auf den tatsächlichen (negierten) Wert gesetzt – dieser kann dann `TRUE` sein.

   Der Kommentar im Netzwerk fasst dies zusammen:

   „ohne die Linie INITO -> REQ ist der Ausgang Q1 beim Start FALSE. mit der Linie ist er TRUE.“

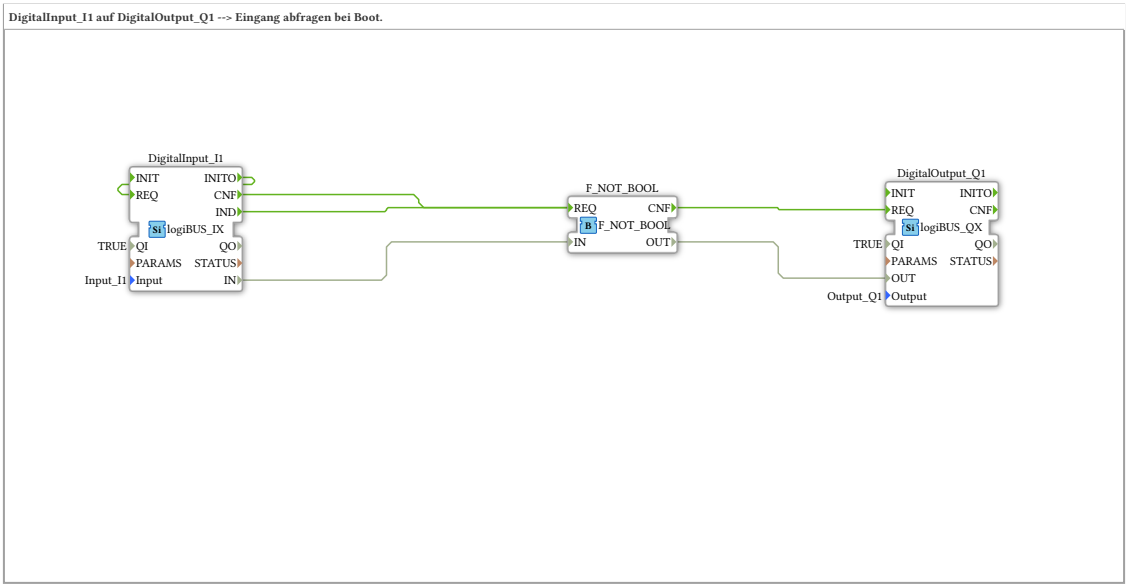
Zusammenfassung

Die Übung demonstriert die grundlegende Verwendung von digitalen Ein- und Ausgangsbausteinen in Kombination mit einer logischen Negation. Der Fokus liegt auf dem Verständnis der ereignisgesteuerten Ablaufsteuerung (Ereigniskette) sowie auf dem Initialisierungsverhalten beim Systemstart. Durch die Rückkopplung des `INITO`-Ereignisses wird sichergestellt, dass der Ausgang bereits beim Booten einen korrekten (negierten) Wert erhält. Dies ist ein typisches Beispiel für die Anwendung von Initialisierungsereignissen in der 4diac-IDE.

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

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

2.2.1.6 Uebung_001c3: DigitalInput_I1 auf DigitalOutput_Q1 –> Eingang abfragen bei Boot



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert das Auslesen eines digitalen Eingangs (Input I1) und die direkte Ausgabe auf einen digitalen Ausgang (Output Q1) beim Hochfahren der Steuerung. Besonderheit ist die Nutzung einer Negation des Eingangssignals sowie einer speziellen Ereignisverbindung, die den Ausgang beim Booten auf TRUE setzt.

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_I1** (Typ: `logiBUS::io::DI::logiBUS_IX`)
  - Parameter: `QI = TRUE`, `Input = Input_I1`
  - Stellt den physikalischen digitalen Eingang I1 bereit. Die Datenbereitstellung erfolgt über den Ereignisausgang `IND` (bei neuer gültiger Eingangsinformation) und `INITO` (Initialisierung beim Start).
- **F_NOT_BOOL** (Typ: `iec61131::bitwiseOperators::F_NOT_BOOL`)
  - Parameter: keine
  - Führt eine logische Negation (NOT) des Booleschen Eingangssignals durch.
- **DigitalOutput_Q1** (Typ: `logiBUS::io::DQ::logiBUS_QX`)
  - Parameter: `QI = TRUE`, `Output = Output_Q1`
  - Steuert den physikalischen digitalen Ausgang Q1. Der Ausgangswert wird nur bei einem REQ-Ereignis übernommen.

Programmablauf und Verbindungen

Der Ablauf wird durch die Ereignis- und Datenverbindungen im SubApp-Netzwerk definiert:

1. **Initialisierung**
   - Beim Booten sendet `DigitalInput_I1` das Ereignis `INITO` an seinen eigenen `REQ`-Eingang. Dies bewirkt ein einmaliges Einlesen des Eingangs direkt nach dem Start.
2. **Eingangslesen und Negation**
   - Jedes Mal, wenn ein neuer Wert am Eingang anliegt, sendet `DigitalInput_I1` das Ereignis `IND`.
   - Das Ereignis `IND` (sowie `CNF`) ist mit dem `REQ`-Eingang von `F_NOT_BOOL` verbunden.
   - Gleichzeitig wird der Datenwert `IN` (vom Eingang) auf den `IN`-Eingang von `F_NOT_BOOL` übertragen. **Wichtig:** Diese Datenverbindung ist mit der Eigenschaft `Negated = true` versehen, was eine Negation auf Verbindungsebene bewirkt. Dadurch wird der Eingangswert bereits vor der NOT-Operation invertiert.
3. **Ausgabe**

- Nach der Berechnung sendet `F_NOT_BOOL` das Ereignis `CNF` an den `REQ`-Eingang von `DigitalOutput_Q1`.
- Der negierte Datenwert `OUT` von `F_NOT_BOOL` wird auf den Dateneingang `OUT` des Ausgangsbausteins gelegt.
- Durch die erste Initialisierung (`INITO -> REQ`) ist der Ausgang nach dem Boot sofort aktiv (TRUE) – ohne diese Verbindung wäre er FALSE.

Besonderheiten:

- Die Kombination aus Negation auf der Datenverbindung und dem NOT-Baustein führt zu einer doppelten Negation, wodurch der Eingangswert unverändert am Ausgang anliegt.
- Die Verbindung `INITO -> REQ` stellt sicher, dass der Ausgang beim Start einen definierten Zustand (TRUE) annimmt, auch wenn noch kein gültiger Eingangswert vorliegt.

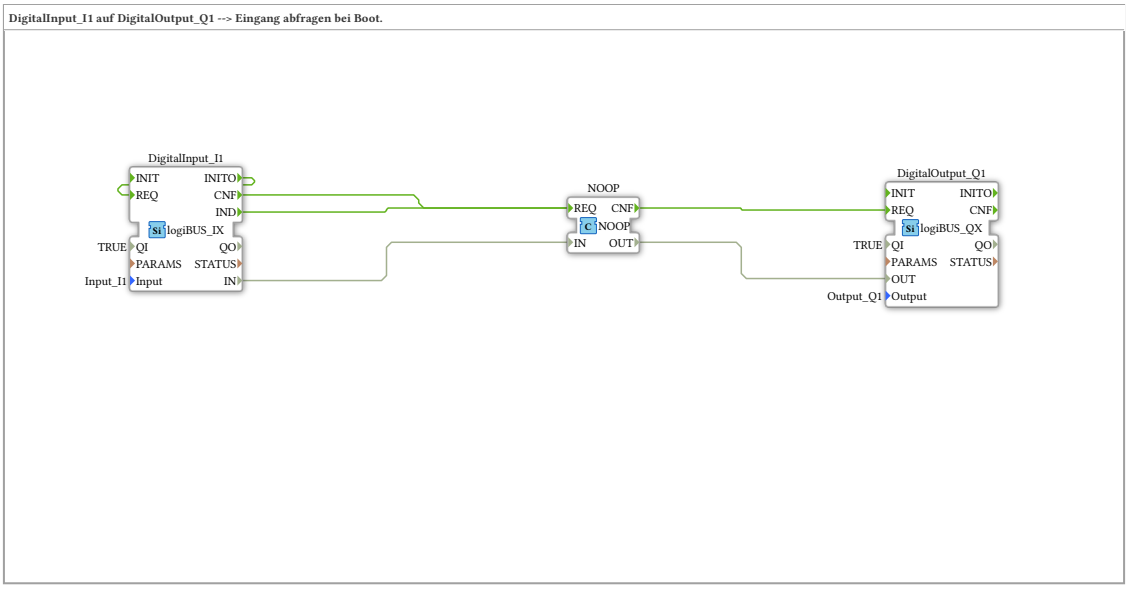
Zusammenfassung

Die Übung zeigt die grundlegende Verknüpfung eines digitalen Eingangs mit einem Ausgang unter 4diac. Durch die Verwendung von Negation und Initialisierungsereignissen wird das Verhalten beim Hochfahren gesteuert. Der Lernende versteht, wie Ereignis- und Datenflüsse in einer IEC 61499-Anwendung aufgebaut werden und wie man mit Negation auf Verbindungsebene arbeitet.

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

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

2.2.1.7 Uebung_001c4: DigitalInput_I1 auf DigitalOutput_Q1 –> Eingang abfragen bei Boot



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert die grundlegende Verwendung eines digitalen Eingangs und eines digitalen Ausgangs auf einem logiBUS-System. Der Eingang **Input_I1** wird beim Systemstart (Boot) abgefragt und sein Zustand direkt auf den Ausgang **Output_Q1** übertragen. Die Übung zeigt, wie durch die Initialisierungs-Eventverbindung (INITO → REQ) sichergestellt wird, dass der Ausgang bereits beim Hochfahren den korrekten Wert annimmt. Zudem wird der **NOOP**-Baustein als einfacher Durchschleif-Baustein eingesetzt, um die Event- und Datenpfade zu verbinden.

Verwendete Funktionsbausteine (FBs)

In dieser Übung werden drei vordefinierte Funktionsbausteine direkt im Netzwerk der SubApp eingesetzt. Es gibt keine weiteren Sub-Bausteine.

- **DigitalInput_I1** (Typ: `logiBUS::io::DI::logiBUS_IX`)
  - **Parameter**:
    - `QI` = `TRUE` (Freigabe des Bausteins)
    - `Input` = `Input_I1` (physikalischer Eingangskanal)
  - **Funktionsweise**: Dieser Baustein liest den Zustand des digitalen Eingangs `Input_I1` ein. Er verfügt über die Ereignisausgänge `IND` (Indikation bei Datenwechsel), `INITO` (Initialisierungsbestätigung) und `CNF` (Bestätigung nach einer Leseanforderung). Der gelesene Wert wird am Datenausgang `IN` bereitgestellt.
- **DigitalOutput_Q1** (Typ: `logiBUS::io::DQ::logiBUS_QX`)
  - **Parameter**:
    - `QI` = `TRUE` (Freigabe des Bausteins)
    - `Output` = `Output_Q1` (physikalischer Ausgangskanal)
  - **Funktionsweise**: Dieser Baustein setzt den digitalen Ausgang `Output_Q1` auf den Wert, der am Dateneingang `OUT` anliegt, sobald er über den Ereigniseingang `REQ` getriggert wird. Der Ausgang wird bei jedem REQ-Ereignis aktualisiert.
- **NOOP** (Typ: `iec61131::bitwiseOperators::NOOP`)
  - **Parameter**: keine
  - **Funktionsweise**: Der NOOP-Baustein ist ein „No Operation"-Baustein, der die an seinem Eingang anliegenden Daten unverändert an den Ausgang weiterleitet. Er dient hier als reiner Durchschleif-Baustein für sowohl Ereignisse als auch Daten. Er kann auch als Gegenstück zu einem `E_TRIG`-Baustein verwendet werden, wenn eine reine Weitergabe ohne Triggerflankenerkennung gewünscht ist.

Programmablauf und Verbindungen

Der Ablauf der Übung ist wie folgt:

1. **Initialisierung**: Beim Start des Systems sendet der Baustein `DigitalInput_I1` nach erfolgreicher Initialisierung ein Ereignis an seinem Ausgang `INITO`. Dieses Ereignis wird auf seinen eigenen Ereigniseingang `REQ` zurückgeführt (Selbsttriggerung). Dadurch wird eine erste Leseanforderung des Eingangs ausgelöst, bevor das eigentliche zyklische Verhalten beginnt. Ohne diese Verbindung wäre der Ausgang `Q1` beim Start **FALSE**, mit der Verbindung ist er **TRUE** (sofern der Eingang eingeschaltet ist).
2. **Zyklisches Lesen**: Nach der Leseanforderung bestätigt `DigitalInput_I1` den Vorgang mit einem `CNF`-Ereignis. Gleichzeitig liefert es bei jeder Änderung des Eingangssignals ein `IND`-Ereignis. Beide Ereignisse (`CNF` und `IND`) werden zum Ereigniseingang `REQ` des **NOOP**-Bausteins weitergeleitet.
3. **Durchschleifen von Daten**: Der gelesene Eingangswert (Datenausgang `IN` des DigitalInput_I1) wird auf den Dateneingang `IN` des NOOP-Bausteins geführt. Der NOOP leitet diesen Wert unverändert an seinen Datenausgang `OUT` weiter.
4. **Ausgang setzen**: Sobald NOOP ein Ereignis an seinem `REQ`-Eingang erhält (von `DigitalInput_I1`), sendet er ein Bestätigungsereignis an seinem Ausgang `CNF`. Dieses `CNF`-Ereignis triggert den Ereigniseingang `REQ` des **DigitalOutput_Q1**-Bausteins. Gleichzeitig liegt der durchgeschleifte Datenwert von NOOP am Dateneingang `OUT` des Ausgangsbausteins an. Daraufhin setzt `DigitalOutput_Q1` den physikalischen Ausgang `Output_Q1` auf den entsprechenden Wert.

Visualisierung der Verbindungen

Die folgende Tabelle zeigt die wesentlichen Verbindungen im Netzwerk:

Von	Nach	Typ
`DigitalInput_I1.INITO`	`DigitalInput_I1.REQ`	Ereignis
`DigitalInput_I1.IND`	`NOOP.REQ`	Ereignis
`DigitalInput_I1.CNF`	`NOOP.REQ`	Ereignis
`NOOP.CNF`	`DigitalOutput_Q1.REQ`	Ereignis
`DigitalInput_I1.IN`	`NOOP.IN`	Daten

NOOP.OUT	DigitalOutput_Q1.OUT	Daten

Lernziele

- Verständnis der Initialisierungssequenz (INITO) und deren Auswirkung auf Ausgangswerte beim Start.
- Umgang mit logiBUS-Eingangs- und Ausgangsbausteinen in 4diac.
- Einsatz des NOOP-Bausteins als reiner Durchschleif-Baustein zur Verbindung von Ereignis- und Datenpfaden.
- Grundlegendes Verständnis von Event-gesteuerten Abläufen in IEC 61499.

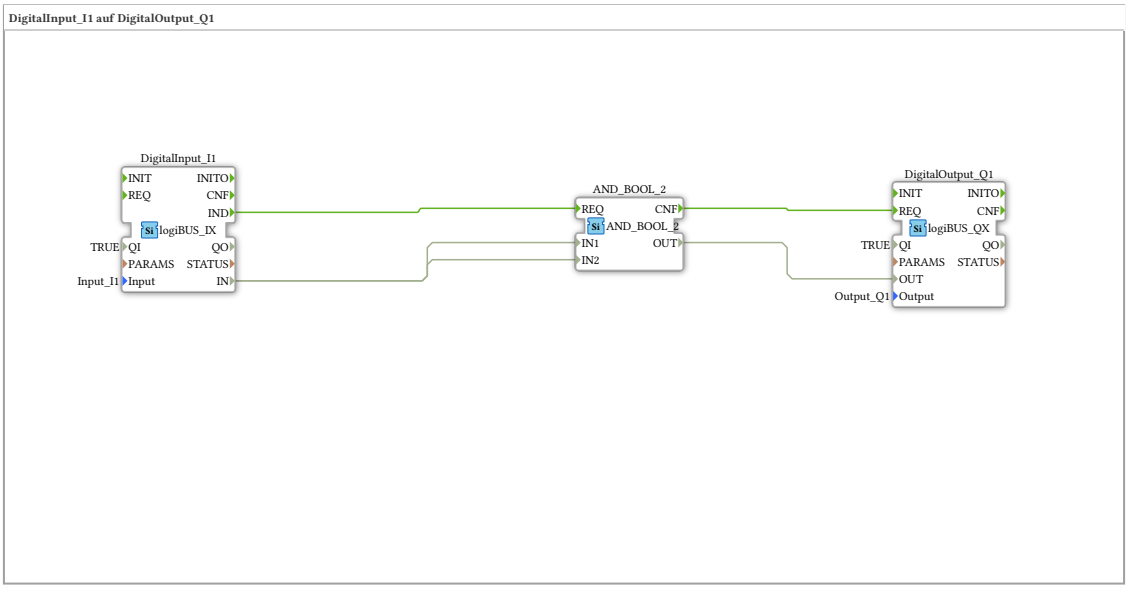
Zusammenfassung

Die Übung **Uebung_001c4** zeigt einen einfachen Anwendungsfall: Ein digitaler Eingang wird direkt auf einen digitalen Ausgang abgebildet. Durch die clever genutzte Selbsttriggerung der Initialisierung wird sichergestellt, dass der Ausgang bereits beim Booten den korrekten Zustand annimmt. Der NOOP-Baustein fungiert als universeller Verbindungsbaustein, der sowohl Ereignisse als auch Daten unverändert weiterleitet. Die Übung eignet sich für Einsteiger, die die Grundlagen der Ereignis- und Datenverknüpfung in 4diac kennenlernen möchten.

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

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

2.2.1.8 Uebung_001d: DigitalInput_I1 auf DigitalOutput_Q1



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert die grundlegende Verknüpfung eines digitalen Eingangs (Input_I1) mit einem digitalen Ausgang (Output_Q1) unter Verwendung einer logischen UND-Verknüpfung. Ziel ist es, das Signal des Eingangs direkt auf den Ausgang zu schalten – hierbei wird das UND-Gatter genutzt, um die Funktionsweise von Ereignis- und Datenflüssen innerhalb der 4diac-IDE kennenzulernen.

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_I1** (Typ: `logiBUS::io::DI::logiBUS_IX`)
  - Parameter:
    - `QI` = `TRUE`
    - `Input` = `Input_I1`
  - Ereignisausgang: `IND` (zeigt an, dass ein neuer Eingangswert vorliegt)
  - Datenausgang: `IN` (aktueller digitaler Wert)

- **AND_2** (Typ: `iec61131::bitwiseOperators::AND_2_BOOL`)
  - Parameter: keine
  - Ereigniseingang: `REQ` (startet die Berechnung)
  - Ereignisausgang: `CNF` (bestätigt die Ausführung)
  - Dateneingänge: `IN1`, `IN2` (zwei boolesche Eingänge)
  - Datenausgang: `OUT` (Ergebnis der UND-Verknüpfung)
- **DigitalOutput_Q1** (Typ: `logiBUS::io::DQ::logiBUS_QX`)
  - Parameter:
    - `QI` = `TRUE`
    - `Output` = `Output_Q1`
  - Ereigniseingang: `REQ` (übernimmt den Ausgangswert)
  - Dateneingang: `OUT` (Wert, der auf den physikalischen Ausgang gesetzt wird)

Programmablauf und Verbindungen

1. Der **DigitalInput_I1** erfasst den Wert des physikalischen Eingangs `Input_I1`. Bei einer Änderung wird das Ereignis `IND` ausgelöst.
2. Dieses Ereignis wird über die Ereignisverbindung an den **AND_2**-Baustein gesendet (an dessen Ereigniseingang `REQ`).
3. Gleichzeitig wird der Datenwert `IN` des Eingangsbausteins über zwei parallele **Datenverbindungen** an die Dateneingänge `IN1` und `IN2` des AND_2-Bausteins weitergegeben.
4. Der **AND_2**-Baustein berechnet die logische UND-Verknüpfung der beiden identischen Signale: `OUT = IN1 AND IN2 = IN (da beide Eingänge gleich sind)`.
5. Nach der Berechnung wird das Ereignis `CNF` ausgelöst, das an den **DigitalOutput_Q1**-Baustein (Ereigniseingang `REQ`) weitergeleitet wird.
6. Der Datenwert `OUT` des AND_2-Bausteins wird an den Dateneingang `OUT` des Ausgangsbausteins übergeben. Dadurch wird der physikalische Ausgang `Output_Q1` auf den gleichen Wert wie der Eingang `Input_I1` gesetzt.

Ergebnis: Das Signal des digitalen Eingangs wird unverändert auf den digitalen Ausgang durchgeschaltet – die UND-Verknüpfung zweier gleicher Signale ändert den Wert nicht.

Zusammenfassung

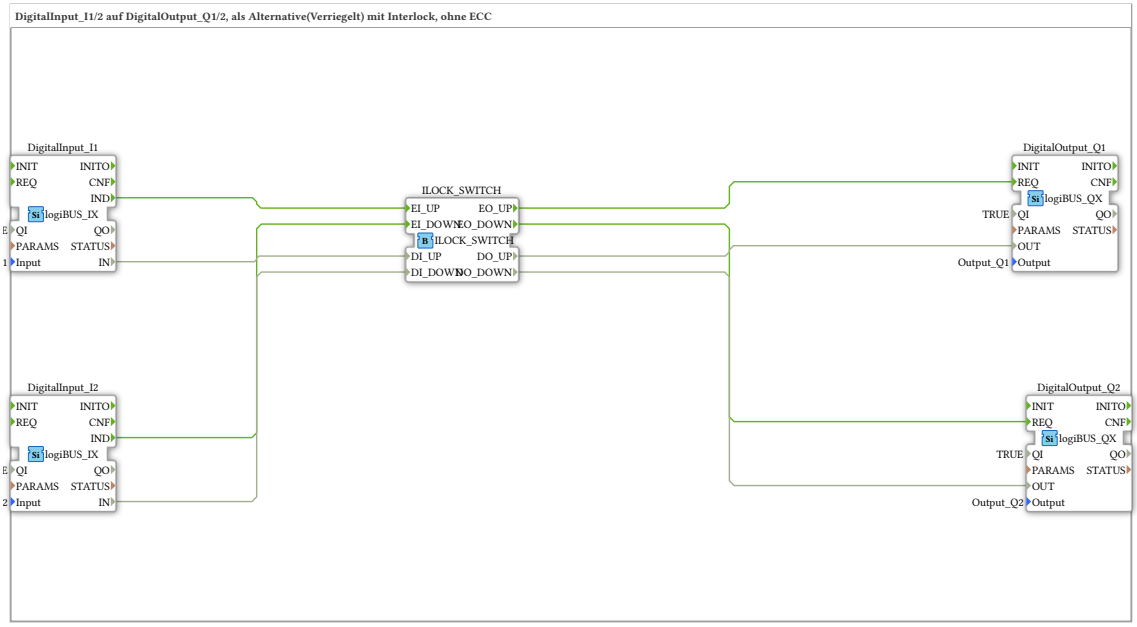
Die Übung führt in die Grundlagen der Ereignis- und Datenverbindungen in der 4diac-IDE ein. Obwohl die UND-Verknüpfung in diesem Fall funktional überflüssig ist, wird das Zusammenspiel zwischen Sensor (Eingang), Logik (UND-Gatter) und Aktor (Ausgang) veranschaulicht. Sie lernen, wie Sie einfache Steuerungsaufgaben durch Verschaltung von Funktionsbausteinen abbilden können.

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

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

2.2.1.9 Uebung_001d2: DigitalInput_I1/2 auf DigitalOutput_Q1/2, als Alternative(Verriegelt) mit Interlock, ohne ECC

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_001d2 (DigitalInput_I1/2 auf DigitalOutput_Q1/2, als Alternative(Verriegelt) mit Interlock, ohne ECC).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **DigitalInput_I1/2 auf DigitalOutput_Q1/2, als Alternative(Verriegelt) mit Interlock, ohne ECC**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_001d2.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IX.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
- **DigitalInput_I2**: Instanz des Typs logiBUS::io::DI::logiBUS_IX.
  - Parameter QI = TRUE
  - Parameter Input = Input_I2
- **ILOCK_SWITCH**: Instanz des Typs logiBUS::signalprocessing::interlock::ILOCK_SWITCH.
- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QX.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
- **DigitalOutput_Q2**: Instanz des Typs logiBUS::io::DQ::logiBUS_QX.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q2

Verbindungen und Schnittstellen

Ereignisverbindungen:

- DigitalInput_I1.IND -> ILOCK_SWITCH.EI_UP
- DigitalInput_I2.IND -> ILOCK_SWITCH.EI_DOWN
- ILOCK_SWITCH.EO_UP -> DigitalOutput_Q1.REQ
- ILOCK_SWITCH.EO_DOWN -> DigitalOutput_Q2.REQ

Datenverbindungen:

- DigitalInput_I1.IN -> ILOCK_SWITCH.DI_UP
- DigitalInput_I2.IN -> ILOCK_SWITCH.DI_DOWN
- ILOCK_SWITCH.DO_UP -> DigitalOutput_Q1.OUT
- ILOCK_SWITCH.DO_DOWN -> DigitalOutput_Q2.OUT

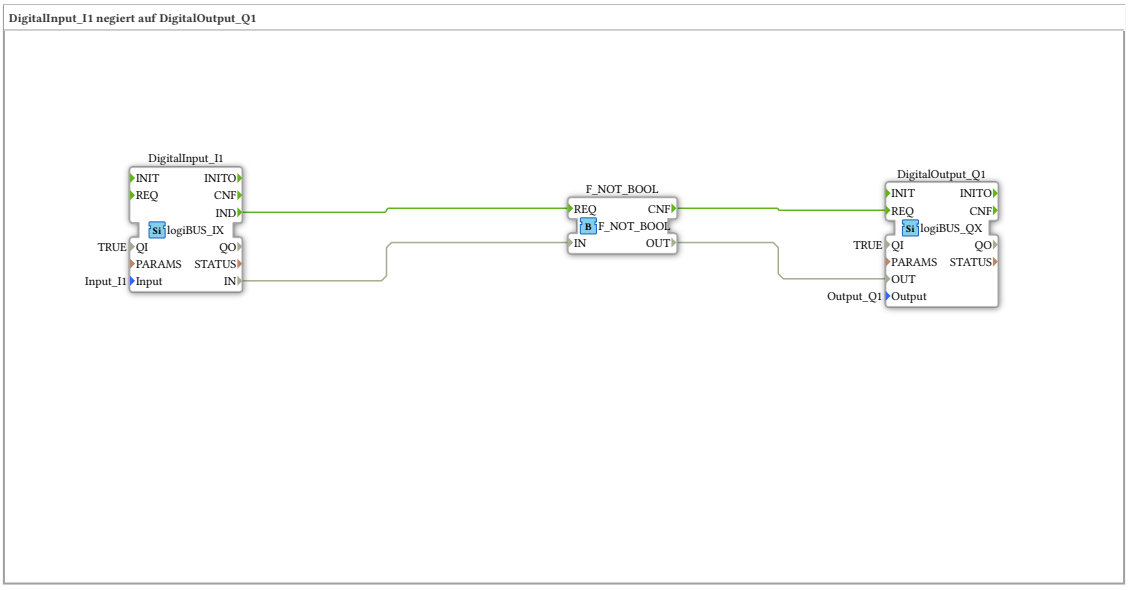
Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_001d2 demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.2.1.10 Uebung_001e: DigitalInput_I1 negiert auf DigitalOutput_Q1



Hinweis: Diese Übung verfügt über kein separates Bild.

Einleitung

Diese Übung demonstriert die grundlegende Negation eines digitalen Eingangssignals. Der digitale Eingang **Input_I1** (Pin I1) wird ausgelesen, logisch negiert und auf den digitalen Ausgang **Output_Q1** (Pin Q1) ausgegeben. Dadurch wird bei aktivem Eingang der Ausgang deaktiviert und umgekehrt.

Die Übung eignet sich für Einsteiger in die industrielle Automatisierung mit IEC 61499 und dient dem Verständnis von:

- Ein- und Ausgabe-Funktionsbausteinen der logiBUS-Bibliothek
- Logischer Negation mittels IEC 61131-Bitoperatoren
- Ereignisgesteuerter Datenverarbeitung

Schwierigkeitsgrad: Einfach **Benötigte Vorkenntnisse:** Grundlegende Kenntnisse der 4diac-IDE und des IEC 61499-Modells

---

Verwendete Funktionsbausteine (FBs)

DigitalInput_I1 (`logiBUS::io::DI::logiBUS_IX`)

- **Typ:** `logiBUS::io::DI::logiBUS_IX`
- **Parameter:**
  - `QI` = `TRUE` (Qualifier – immer aktiv)
  - `Input` = `Input_I1` (physikalischer Pin I1)
- **Ereignisausgang:** `IND` – wird ausgelöst, sobald der Eingangswert aktualisiert wird
- **Datenausgang:** `IN` – der aktuelle digitale Zustand (BOOL) des Pins
- **Funktionsweise:** Liest den digitalen Zustand des Eingangs I1 und stellt ihn am Ausgang `IN` zur Verfügung. Ein eingehendes Ereignis (z. B. über das Netzwerk) aktiviert die Auslese.

F_NOT_BOOL (`iec61131::bitwiseOperators::F_NOT_BOOL`)

- **Typ:** `iec61131::bitwiseOperators::F_NOT_BOOL`
- **Parameter:** Keine
- **Ereigniseingang:** `REQ` – startet die Negation
- **Ereignisausgang:** `CNF` – bestätigt die fertige Negation
- **Dateneingang:** `IN` (BOOL) – zu negierender Wert
- **Datenausgang:** `OUT` (BOOL) – negierter Wert
- **Funktionsweise:** Führt eine logische NOT-Operation auf den booleschen Eingangswert aus. Bei `IN = TRUE` wird `OUT = FALSE` und umgekehrt.

DigitalOutput_Q1 (`logiBUS::io::DQ::logiBUS_QX`)

- **Typ:** `logiBUS::io::DQ::logiBUS_QX`
- **Parameter:**
  - `QI` = `TRUE` (Qualifier – immer aktiv)
  - `Output` = `Output_Q1` (physikalischer Pin Q1)
- **Ereigniseingang:** `REQ` – löst das Setzen des Ausgangs aus
- **Dateneingang:** `OUT` (BOOL) – gewünschter Zustand des Ausgangs
- **Funktionsweise:** Setzt den digitalen Ausgang Q1 auf den am Eingang `OUT` anliegenden Wert, sobald ein Ereignis am `REQ`-Eingang eintrifft.

Programmablauf und Verbindungen

Der Ablauf ist strikt ereignisgesteuert:

1. **Eingangsereignis:** Der Funktionsbaustein `DigitalInput_I1` erzeugt ein Ereignis an seinem Ausgang `IND`, sobald der Eingang I1 einen neuen Wert liefert (z. B. durch eine zyklische Abfrage oder externe Änderung).
2. **Negation starten:** Dieses `IND`-Ereignis wird über eine **Eventverbindung** an den Ereigniseingang `REQ` von `F_NOT_BOOL` weitergeleitet. Gleichzeitig wird der aktuelle Datenwert von `DigitalInput_I1.IN` über eine **Datenverbindung** an den Eingang `F_NOT_BOOL.IN` übergeben.
3. **Negation ausführen:** `F_NOT_BOOL` berechnet den negierten Wert und gibt ihn an seinem Ausgang `OUT` aus. Sobald die Berechnung abgeschlossen ist, wird ein Ereignis am Ausgang `CNF` erzeugt.
4. **Ausgang setzen:** Das `CNF`-Ereignis wird über eine weitere **Eventverbindung** an den Ereigniseingang `REQ` von `DigitalOutput_Q1` gesendet. Parallel dazu wird der negierte Datenwert von `F_NOT_BOOL.OUT` über eine **Datenverbindung** an den Dateneingang `DigitalOutput_Q1.OUT` angelegt. Dadurch wird der Ausgang Q1 mit dem negierten Wert aktualisiert.

Zusammenfassung der Verbindungen:

- Event: `DigitalInput_I1.IND` ⟶ `F_NOT_BOOL.REQ` ⟶ `F_NOT_BOOL.CNF` ⟶ `DigitalOutput_Q1.REQ`
- Data: `DigitalInput_I1.IN` ⟶ `F_NOT_BOOL.IN` ⟶ `F_NOT_BOOL.OUT` ⟶ `DigitalOutput_Q1.OUT`

Die Übung kann in 4diac durch Starten der Applikation (z. B. mit einem Simulationslauf) ausgeführt werden. Beobachten Sie das Verhalten: Der Ausgang Q1 ist genau dann aktiv, wenn der Eingang I1 nicht aktiv ist (und umgekehrt).

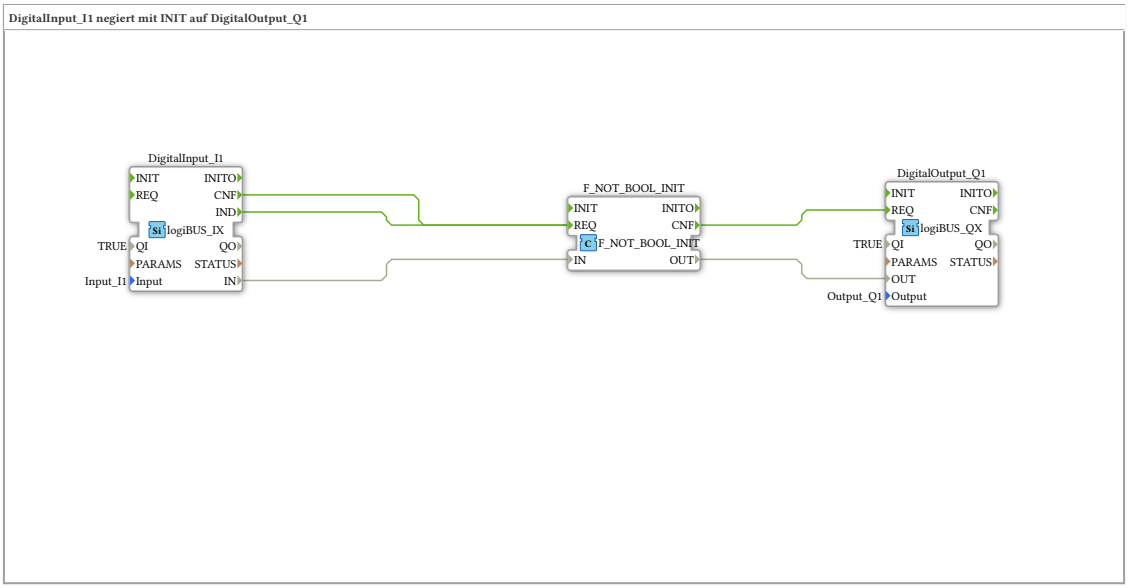
Zusammenfassung

Die Übung **Uebung_001e** realisiert eine einfache Negation eines digitalen Eingangs mithilfe einer Kaskade von drei Funktionsbausteinen. Sie vermittelt die Grundprinzipien der ereignisgesteuerten Datenverarbeitung nach IEC 61499 sowie den Umgang mit logiBUS-Ein-/Ausgabe-Bausteinen und IEC 61131-Bitoperatoren. Nach erfolgreicher Durchführung verstehen Sie, wie Eingangsdaten in Echtzeit verarbeitet und auf Ausgänge geschrieben werden. Dies bildet die Basis für komplexere Automatisierungsanwendungen.

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

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

2.2.1.11 Uebung_001f: DigitalInput_I1 negiert mit INIT auf DigitalOutput_Q1



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert die Negation eines digitalen Eingangssignals mithilfe des Funktionsbausteins `F_NOT_BOOL_INIT`. Der digitale Eingang `Input_I1` wird gelesen, invertiert und auf den digitalen Ausgang `Output_Q1` geschrieben. Dabei wird deutlich, dass der Negationsbaustein bereits beim Systemstart (BOOT) einen definierten Wert ausgibt, auch wenn der Eingang zu diesem Zeitpunkt noch nicht gelesen wird.

Lernziel: Verständnis der Verknüpfung von Ein-/Ausgangsbausteinen mit Logik-FBs sowie der Initialisierung von Negationsbausteinen.

Verwendete Funktionsbausteine (FBs)

Die Übung besteht aus drei direkt im Netzwerk platzierten Funktionsbausteinen. Es werden keine Sub-Bausteine verwendet.

- **DigitalInput_I1**
  - **Typ:** `logiBUS::io::DI::logiBUS_IX`
  - **Parameter:** `QI = TRUE`, `Input = Input_I1`
  - **Funktion:** Liest den digitalen Eingang `Input_I1` und stellt den Wert am Datenausgang `IN` sowie die Ereignisse `IND` (bei steigender Flanke) und `CNF` (bei fallender Flanke) bereit.
- **F_NOT_BOOL_INIT**
  - **Typ:** `iec61131::bitwiseOperators::F_NOT_BOOL_INIT`
  - **Parameter:** keine weiteren
  - **Funktion:** Negiert den anliegenden booleschen Wert am Dateneingang `IN` und gibt das Ergebnis am Datenausgang `OUT` aus. Der Baustein verfügt über einen eingebauten INIT-Mechanismus, der beim Start des Systems einen definierten Ausgangswert liefert.
- **DigitalOutput_Q1**
  - **Typ:** `logiBUS::io::DQ::logiBUS_QX`
  - **Parameter:** `QI = TRUE`, `Output = Output_Q1`
  - **Funktion:** Schreibt den am Dateneingang `OUT` anliegenden booleschen Wert auf den Ausgang `Output_Q1`, sobald das Ereignis `REQ` empfangen wird.

Programmablauf und Verbindungen

Das Netzwerk der Übung arbeitet ereignisgesteuert:

1. **Ereignisverbindungen:**

- Die Ausgangsereignisse `IND` und `CNF` des DigitalInput_I1 werden beide auf den Ereigniseingang `REQ` des F_NOT_BOOL_INIT geführt.
- Das Bestätigungsereignis `CNF` des F_NOT_BOOL_INIT wird auf den Ereigniseingang `REQ` des DigitalOutput_Q1 geschaltet.

2. **Datenverbindungen:**

- Der Datenausgang `IN` von DigitalInput_I1 ist mit dem Dateneingang `IN` von F_NOT_BOOL_INIT verbunden.
- Der Datenausgang `OUT` von F_NOT_BOOL_INIT ist mit dem Dateneingang `OUT` von DigitalOutput_Q1 verbunden.

Ablauf:

- Sobald sich der Zustand des digitalen Eingangs `Input_I1` ändert (steigende oder fallende Flanke), erzeugt DigitalInput_I1 das entsprechende Ereignis.
- Dieses Ereignis triggert den Negationsbaustein, der den aktuellen Eingangswert invertiert und nach Abschluss der Berechnung das Ereignis `CNF` ausgibt.
- Dadurch wird der Ausgangsbaustein getriggert, den negierten Wert auf `Output_Q1` zu schreiben.

Hinweis: Da der Negationsbaustein `F_NOT_BOOL_INIT` einen INIT-Mechanismus besitzt, wird bereits beim Hochfahren des Systems (ohne vorherige Eingangsänderung) ein initialer negierter Wert am Ausgang anliegen – auch wenn der Eingang zu diesem Zeitpunkt noch nicht gelesen wurde. Dies ist durch den Kommentar im Netzwerk verdeutlicht.

Schwierigkeitsgrad: Einfach **Vorkenntnisse:** Grundlegende Kenntnisse der Ereignis- und Datenflussmodellierung in 4diac sowie der Ein-/Ausgangskonfiguration über logiBUS.

Zusammenfassung

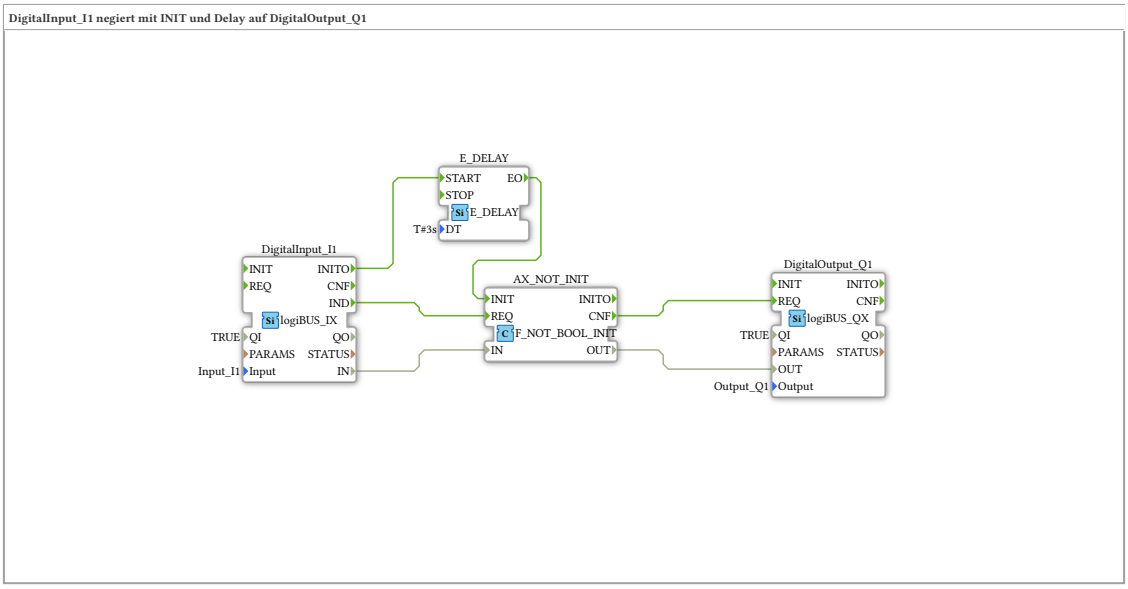
Die Übung `Uebung_001f` realisiert eine einfache Negation eines digitalen Eingangssignals auf einen digitalen Ausgang. Sie zeigt die grundlegende Verbindung von Hardware-nahen Funktionsbausteinen mit einem logischen Negationsbaustein und verdeutlicht das initiale Verhalten des Negations-FBs beim Systemstart. Die Implementierung erfolgt ausschließlich über direkte FB-Verkettung ohne Sub-Bausteine und ist als Einstiegsübung für die ereignisgesteuerte Logik mit 4diac geeignet.

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

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

2.2.1.12 Uebung_001g: DigitalInput_I1 negiert mit INIT und Delay auf DigitalOutput_Q1

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_001g (DigitalInput_I1 negiert mit INIT und Delay auf DigitalOutput_Q1).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **DigitalInput_I1 negiert mit INIT und Delay auf DigitalOutput_Q1**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_001g.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QX.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
- **DigitalInput_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IX.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
- **AX_NOT_INIT**: Instanz des Typs iec61131::booleanOperators::F_NOT_BOOL_INIT.
- **E_DELAY**: Instanz des Typs iec61499::events::E_DELAY.
  - Parameter DT = T#3s

Verbindungen und Schnittstellen

Ereignisverbindungen:

- DigitalInput_I1.INITO -> E_DELAY.START
- E_DELAY.EO -> AX_NOT_INIT.INIT
- AX_NOT_INIT.CNF -> DigitalOutput_Q1.REQ
- DigitalInput_I1.IND -> AX_NOT_INIT.REQ

Datenverbindungen:

- AX_NOT_INIT.OUT -> DigitalOutput_Q1.OUT
- DigitalInput_I1.IN -> AX_NOT_INIT.IN

Hinweise aus dem Modell

obwohl I1 nicht abgefragt wird beim BOOT, wird AX_NOT hier TRUE ausgeben.

Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.

2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_001g demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.2.1.13 Uebung_002: DigitalInput_I1 auf DigitalOutput_Q1/_Q2

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_002`, bei der ein einzelnes digitales Eingangssignal auf zwei verschiedene digitale Ausgänge verteilt wird. Hierbei wird das Konzept des „Fan-Out“ (Vervielfachung) von Verbindungen demonstriert.

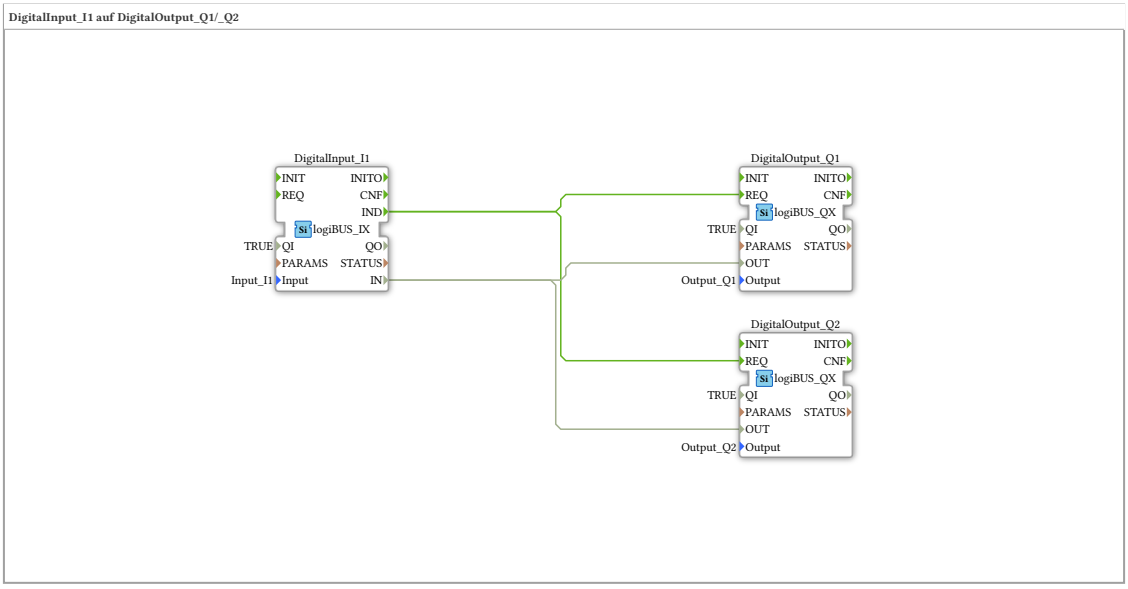
Ziel der Übung

Das Hauptziel dieser Übung ist es, zu zeigen, wie Ereignis- und Datenverbindungen in der IEC 61499 verzweigt werden können. Ein einzelner Quell-Port kann mehrere Ziel-Ports bedienen. Dies ist eine fundamentale Methode, um parallele Aktionen in einer Steuerung auszulösen.

Beschreibung und Komponenten

In der Subapplikation `Uebung_002.SUB` wird ein digitaler Eingang eingelesen und direkt an zwei digitale Ausgänge weitergereicht.

Funktionsbausteine (FBs)



- **`DigitalInput_I1`**: Eine Instanz des Typs `logiBUS_IX`. Dieser Baustein liest den Hardware-Eingang `Input_I1`.
- **`DigitalOutput_Q1` & `DigitalOutput_Q2`**: Instanzen des Typs `logiBUS_QX`. Diese repräsentieren die physischen Ausgänge `Output_Q1` und `Output_Q2`.

Funktionsweise

Die Signalverteilung wird durch das Ziehen von jeweils zwei Verbindungen von der Quelle zu den Zielen erreicht. Der Aufbau in `Uebung_002.SUB` ist wie folgt definiert:

```
<EventConnections>
    <Connection Source="DigitalInput_I1.IND" Destination="DigitalOutput_Q1.REQ"/>
```

```
    <Connection Source="DigitalInput_I1.IND" Destination="DigitalOutput_Q2.REQ"/>
</EventConnections>
<DataConnections>
    <Connection Source="DigitalInput_I1.IN" Destination="DigitalOutput_Q1.OUT"/>
    <Connection Source="DigitalInput_I1.IN" Destination="DigitalOutput_Q2.OUT"/>
</DataConnections>
```

Der Signalweg verläuft dabei in folgenden Schritten:

1. Der Baustein `DigitalInput_I1` detektiert eine Änderung am physischen Eingang.
2. Ein Ereignis wird am Port `IND` ausgelöst und an **beide** Zielbausteine (`Q1` und `Q2`) gesendet.
3. Zeitgleich steht der aktuelle Signalzustand am Port `IN` für beide Bausteine zur Verfügung.
4. Beide Ausgangsbausteine empfangen das Ereignis zeitgleich und schalten ihre jeweiligen Hardware-Ausgänge auf den gelieferten Wert.

Im Ergebnis schalten beide Ausgänge synchron zum Zustand des Eingangs `I1`.

Anwendungsbeispiel

Ein typisches Anwendungsbeispiel ist die **parallele Statusanzeige**: Ein Sensor an einer Maschine (`I1`) soll nicht nur die interne Logik steuern, sondern gleichzeitig eine lokale Kontrollleuchte (`Q1`) und eine Signallampe an einem entfernten Bedienpult (`Q2`) aktivieren. Durch die Verzweigung der Signale wird sichergestellt, dass beide Anzeigen immer den identischen Zustand des Sensors widerspiegeln.

2.2.1.14 Uebung_002a: DigitalInput_I1/_I2 mit AND auf DigitalOutput_Q1



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_002a`. In dieser Übung wird eine klassische UND-Verknüpfung (AND) realisiert, bei der ein digitaler Ausgang nur dann aktiviert wird, wenn zwei digitale Eingänge gleichzeitig den Zustand „Wahr“ (HIGH) führen.

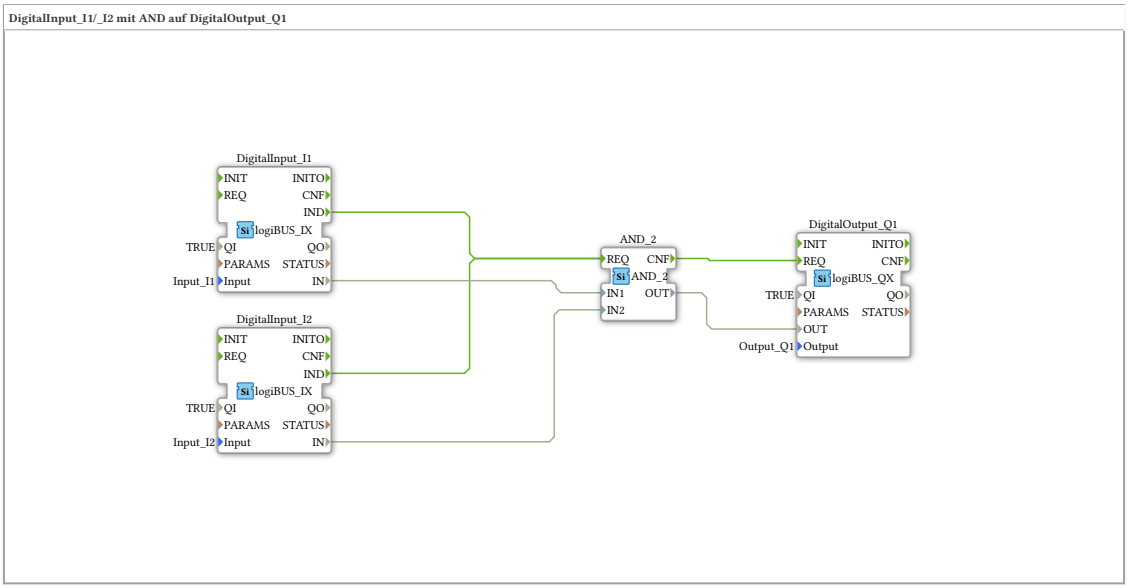
Ziel der Übung

Das Hauptziel dieser Übung ist die Implementierung einer grundlegenden logischen Entscheidungsstruktur. Es wird gezeigt, wie Signale von mehreren Sensoren (Eingängen) kombiniert werden können, um eine Aktion an einem Aktor (Ausgang) auszulösen. Dies ist ein fundamentaler Baustein jeder Steuerungsprogrammierung.

Beschreibung und Komponenten

Die Subapplikation `Uebung_002a.SUB` verknüpft zwei digitale Eingänge über einen Logik-Baustein mit einem digitalen Ausgang.

Funktionsbausteine (FBs)



- **`DigitalInput_I1`** & **`DigitalInput_I2`**: Instanzen des Typs `logiBUS_IX`. Diese repräsentieren die beiden Hardware-Eingänge, die überwacht werden.
- **`AND_2`**: Eine Instanz des Typs `AND_2` (aus der IEC 61131-Bibliothek). Dieser Baustein führt die logische UND-Operation aus. Er besitzt zwei Dateneingänge (`IN1`, `IN2`) und einen Datenausgang (`OUT`). Zur Steuerung benötigt er ein Ereignis am Port `REQ` und quittiert die Berechnung am Port `CNF`.
- **`DigitalOutput_Q1`**: Eine Instanz des Typs `logiBUS_QX`. Dieser Baustein steuert den Hardware-Ausgang `Output_Q1` basierend auf dem Ergebnis der Logik.

Funktionsweise

Die Logik wird durch die Verschaltung von Ereignis- und Datenverbindungen festgelegt. Der Aufbau in `Uebung_002a.SUB` ist wie folgt definiert:

```
<EventConnections>
    <Connection Source="DigitalInput_I1.IND" Destination="AND_2.REQ"/>
    <Connection Source="DigitalInput_I2.IND" Destination="AND_2.REQ"/>
    <Connection Source="AND_2.CNF" Destination="DigitalOutput_Q1.REQ"/>
</EventConnections>
<DataConnections>
    <Connection Source="DigitalInput_I1.IN" Destination="AND_2.IN1"/>
    <Connection Source="DigitalInput_I2.IN" Destination="AND_2.IN2"/>
    <Connection Source="AND_2.OUT" Destination="DigitalOutput_Q1.OUT"/>
</DataConnections>
```

Der Prozess folgt dieser Logik:

1. Ändert sich einer der beiden Eingänge (`I1` oder `I2`), sendet der jeweilige Baustein ein `IND`-Ereignis an den `REQ`-Port des `AND_2`-Bausteins.
2. Der `AND_2`-Baustein liest daraufhin beide Daten-Eingänge (`IN1` und `IN2`) und berechnet das Ergebnis (`IN1 AND IN2`).
3. Nach Abschluss der Berechnung feuert der Logik-Baustein ein `CNF`-Ereignis (Confirmation) ab.
4. Dieses `CNF`-Ereignis erreicht den `REQ`-Port von `DigitalOutput_Q1`, welcher daraufhin das Ergebnis übernimmt und den physischen Ausgang schaltet.

Anwendungsbeispiel

Ein klassisches Anwendungsbeispiel ist die **Sicherheits-Freigabe**: Ein Motor (`Q1`) soll nur starten, wenn sowohl die Schutztür geschlossen ist (`I1`) als auch der Bediener den Start-Taster drückt (`I2`). Nur wenn beide Bedingungen gleichzeitig erfüllt sind (`TRUE`), liefert das UND-Gatter ein Signal zum Einschalten des Motors.

2.2.1.15 Uebung_002a2: DigitalInput_I1/_I2 mit AND (alt) auf DigitalOutput_Q1

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_002a2`. Diese Übung ist funktional identisch mit `Uebung_002a`, demonstriert jedoch die Verwendung des generischen Funktionsbausteins `F_AND` anstelle des typspezifischen `AND_2`.

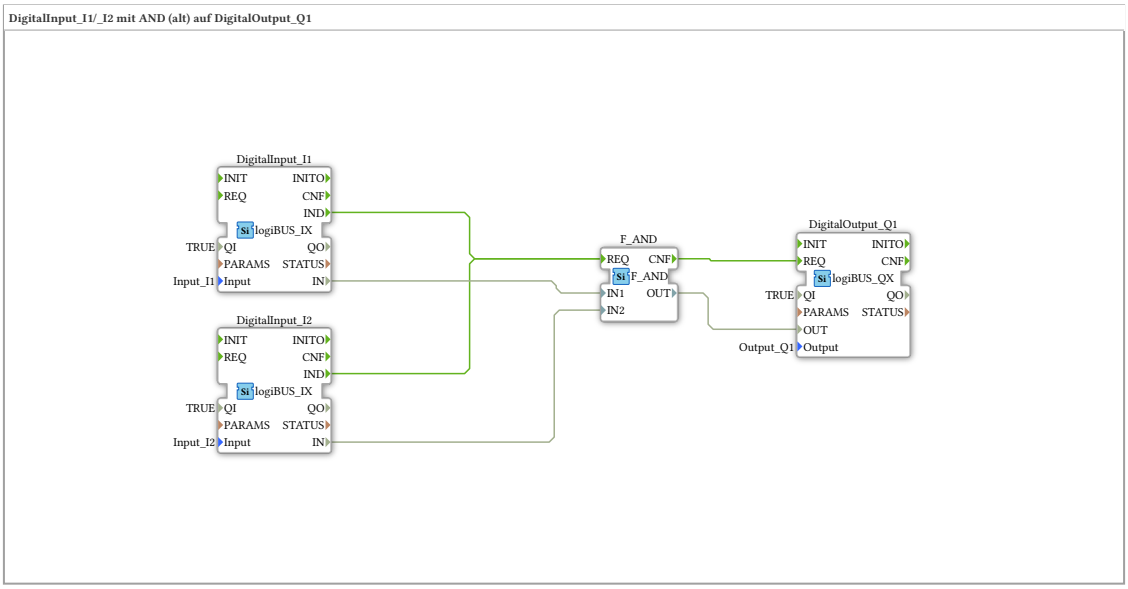
Ziel der Übung

Das Ziel ist es, die Verwendung von generischen Funktionsbausteinen (F-FBs) in der IEC 61499 zu verstehen. Es wird gezeigt, dass unterschiedliche Bausteintypen dieselbe logische Operation (UND) ausführen können, wobei das ereignisbasierte Ausführungsmodell identisch bleibt.

Beschreibung und Komponenten

In der Subapplikation `Uebung_002a2.SUB` werden zwei digitale Eingänge über ein generisches UND-Gatter verknüpft.

Funktionsbausteine (FBs)



- **`DigitalInput_I1` & `DigitalInput_I2`**: Standard-Eingangsbausteine vom Typ `logiBUS_IX`.
- **`F_AND`**: Ein generischer Funktionsbaustein vom Typ `F_AND`. Er berechnet das logische UND seiner Eingänge `IN1` und `IN2`, sobald er ein Ereignis am Eingang `REQ` empfängt, und gibt das Ergebnis am Ausgang `OUT` sowie ein Bestätigungs-Ereignis am Port `CNF` aus.
- **`DigitalOutput_Q1`**: Standard-Ausgangsbaustein vom Typ `logiBUS_QX`.

Funktionsweise

Der Aufbau in `Uebung_002a2.SUB` folgt dem bewährten Muster der Ereigniskette:

```
<EventConnections>
    <Connection Source="DigitalInput_I1.IND" Destination="F_AND.REQ"/>
    <Connection Source="DigitalInput_I2.IND" Destination="F_AND.REQ"/>
    <Connection Source="F_AND.CNF" Destination="DigitalOutput_Q1.REQ"/>
</EventConnections>
<DataConnections>
    <Connection Source="DigitalInput_I1.IN" Destination="F_AND.IN1"/>
    <Connection Source="DigitalInput_I2.IN" Destination="F_AND.IN2"/>
    <Connection Source="F_AND.OUT" Destination="DigitalOutput_Q1.OUT"/>
</DataConnections>
```

Der funktionale Ablauf:

1. Jede Änderung an den Tastern `I1` oder `I2` löst ein `IND`-Event aus.
2. Beide Events sind mit dem `REQ`-Port von `F_AND` verbunden. Das bedeutet: Egal welcher Taster betätigt wird, die Logik wird neu berechnet.
3. `F_AND` ermittelt das Ergebnis.
4. Über das `CNF`-Event wird der Baustein `DigitalOutput_Q1` angewiesen, den Hardware-Ausgang `Q1` zu aktualisieren.

Der Ausgang ist nur dann aktiv, wenn beide Eingänge gleichzeitig den Wert `TRUE` führen.

Anwendungsbeispiel

Zustimmungs-Schaltung: Ein Bediener muss an einem Steuerpult eine Taste drücken (`I1`) und gleichzeitig muss ein zweiter Sensor (`I2`) die Anwesenheit eines Werkstücks bestätigen, damit der Roboterarm (`Q1`) das Werkstück greifen darf.

2.2.1.16 Uebung_002a3: DigitalInput_I1/_I2 mit OR auf DigitalOutput_Q1



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_002a3`. In dieser Übung wird eine logische ODER-Verknüpfung (OR) implementiert, bei der ein digitaler Ausgang aktiviert wird, sobald mindestens einer von zwei digitalen Eingängen den Zustand „Wahr“ (HIGH) einnimmt.

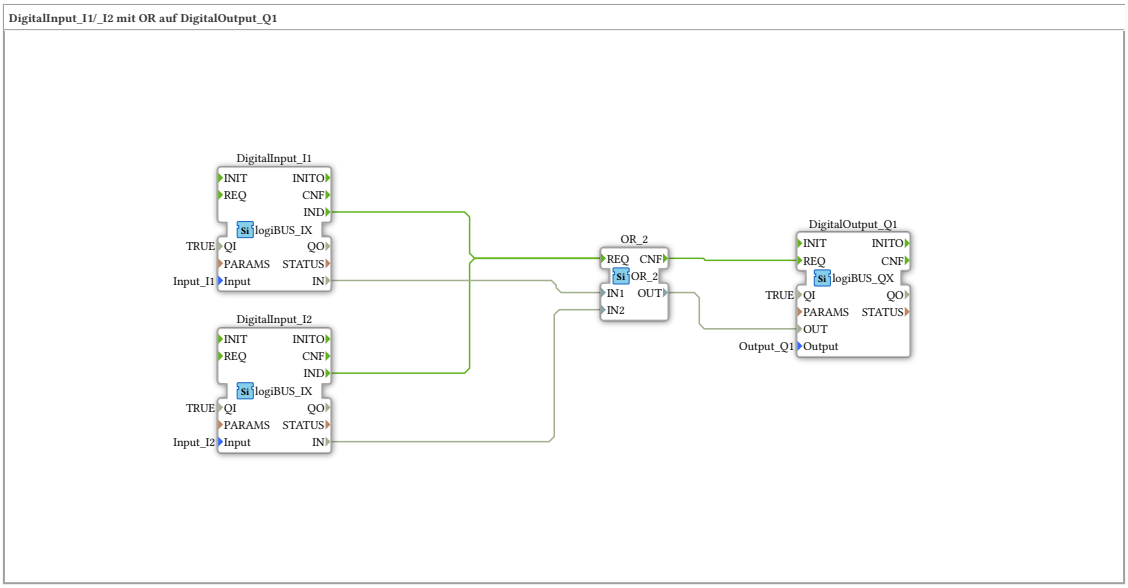
Ziel der Übung

Das Hauptziel dieser Übung ist es, die Funktionsweise einer ODER-Verknüpfung in der Automatisierungstechnik zu demonstrieren. Sie zeigt, wie alternative Bedingungen (Eingänge) genutzt werden können, um dieselbe Aktion (Ausgang) auszulösen. Dies ist eine Standardanforderung für Systeme, die von mehreren Stellen aus bedienbar sein müssen.

Beschreibung und Komponenten

Die Subapplikation `Uebung_002a3.SUB` führt zwei digitale Eingangssignale über einen ODER-Logik-Baustein zusammen.

Funktionsbausteine (FBs)



- **`DigitalInput_I1` & `DigitalInput_I2`**: Instanzen des Typs `logiBUS_IX`. Diese Bausteine erfassen die Zustände der physischen Eingänge `Input_I1` und `Input_I2`.

- **OR_2**: Eine Instanz des Typs OR_2 (aus der IEC 61131-Bibliothek). Dieser Baustein führt die logische ODER-Operation aus. Er besitzt zwei Dateneingänge (IN1, IN2) und einen Datenausgang (OUT). Wie der AND-Baustein reagiert er auf ein Ereignis am Port REQ und quittiert dies mit CNF.
- **DigitalOutput_Q1**: Eine Instanz des Typs logiBUS_QX. Dieser Baustein setzt den physischen Ausgang Output_Q1 basierend auf dem Ergebnis der ODER-Verknüpfung.

Funktionsweise

Die Logik wird durch die Verschaltung von Ereignis- und Datenverbindungen definiert. Der Aufbau in Uebung_002a3.SUB ist wie folgt:

```
<EventConnections>
    <Connection Source="DigitalInput_I1.IND" Destination="OR_2.REQ"/>
    <Connection Source="DigitalInput_I2.IND" Destination="OR_2.REQ"/>
    <Connection Source="OR_2.CNF" Destination="DigitalOutput_Q1.REQ"/>
</EventConnections>
<DataConnections>
    <Connection Source="DigitalInput_I1.IN" Destination="OR_2.IN1"/>
    <Connection Source="DigitalInput_I2.IN" Destination="OR_2.IN2"/>
    <Connection Source="OR_2.OUT" Destination="DigitalOutput_Q1.OUT"/>
</DataConnections>
```

Der Prozess folgt dieser Logik:

1. Jede Änderung an den Tastern I1 oder I2 löst ein IND-Event aus.
2. Beide Events sind mit dem REQ-Port von OR_2 verbunden. Das bedeutet: Egal welcher Taster betätigt wird, die Logik wird neu berechnet.
3. OR_2 prüft die Zustände: Wenn mindestens ein Eingang den Wert TRUE führt, wird der Ausgang OUT ebenfalls TRUE.
4. Über das CNF-Event wird der Baustein DigitalOutput_Q1 angewiesen, den Hardware-Ausgang Q1 zu aktualisieren.

Anwendungsbeispiel

Ein typisches Anwendungsbeispiel ist die **Flurbeleuchtung mit zwei Schaltern**: In einem langen Flur gibt es an beiden Enden einen Schalter (I1 und I2). Die Lampe (Q1) soll leuchten, wenn Schalter 1 betätigt wird ODER wenn Schalter 2 betätigt wird. Diese „Entweder-Oder"-Logik wird durch den OR_2-Baustein abgebildet, sodass man die Beleuchtung von beiden Stellen aus unabhängig voneinander einschalten kann.

2.2.1.17 Uebung_002a4: DigitalInput_I1/_I2 mit AND_BOOL auf DigitalOutput_Q1

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung Uebung_002a4. In dieser Übung wird eine logische UND-Verknüpfung (AND) realisiert, wobei ein digitaler Ausgang nur dann aktiviert wird, wenn zwei digitale Eingänge gleichzeitig den Zustand „Wahr" (HIGH) führen.

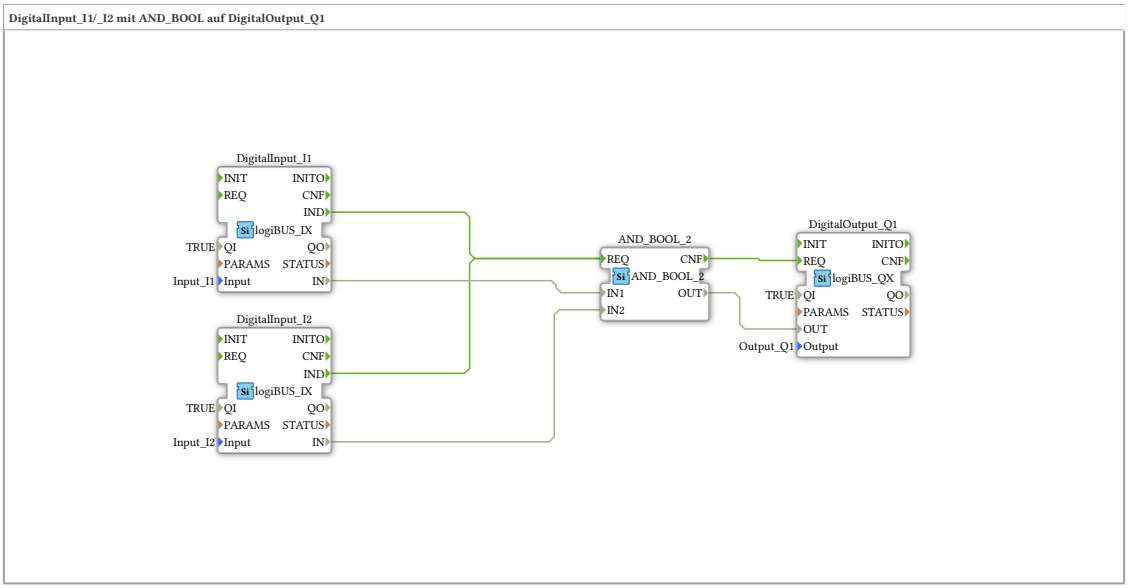
Ziel der Übung

Das Hauptziel dieser Übung ist die Implementierung einer logischen Entscheidungsstruktur unter Verwendung des spezialisierten Typs AND_2_BOOL. Es wird gezeigt, wie Ereignis- und Datenflüsse kombiniert werden, um eine Hardware-Ausgabe basierend auf mehreren Eingangsbedingungen zu steuern.

Beschreibung und Komponenten

Die Subapplikation Uebung_002a4.SUB verknüpft zwei digitale Eingänge über einen Logik-Baustein mit einem digitalen Ausgang.

Funktionsbausteine (FBs)



- **`DigitalInput_I1` & `DigitalInput_I2`**: Instanzen des Typs `logiBUS_IX`. Diese repräsentieren die beiden Hardware-Eingänge, die überwacht werden.
- **`AND_2_BOOL`**: Eine Instanz des Typs `AND_2_BOOL` (aus der IEC 61131-Bibliothek). Dieser Baustein führt die logische UND-Operation speziell für boolesche Werte aus. Er besitzt zwei Dateneingänge (`IN1`, `IN2`) und einen Datenausgang (`OUT`). Wie alle Standard-Logikbausteine reagiert er auf ein Ereignis am Port `REQ` und signalisiert die Fertigstellung am Port `CNF`.
- **`DigitalOutput_Q1`**: Eine Instanz des Typs `logiBUS_QX`. Dieser Baustein steuert den Hardware-Ausgang `Output_Q1`.

Funktionsweise

Die Logik wird durch die Verschaltung der Ereignis- und Datenpfade in der Subapplikation festgelegt. Der Aufbau in `Uebung_002a4.SUB` ist wie folgt definiert:

```
<EventConnections>
    <Connection Source="DigitalInput_I1.IND" Destination="AND_2_BOOL.REQ"/>
    <Connection Source="DigitalInput_I2.IND" Destination="AND_2_BOOL.REQ"/>
    <Connection Source="AND_2_BOOL.CNF" Destination="DigitalOutput_Q1.REQ"/>
</EventConnections>
<DataConnections>
    <Connection Source="DigitalInput_I1.IN" Destination="AND_2_BOOL.IN1"/>
    <Connection Source="DigitalInput_I2.IN" Destination="AND_2_BOOL.IN2"/>
    <Connection Source="AND_2_BOOL.OUT" Destination="DigitalOutput_Q1.OUT"/>
</DataConnections>
```

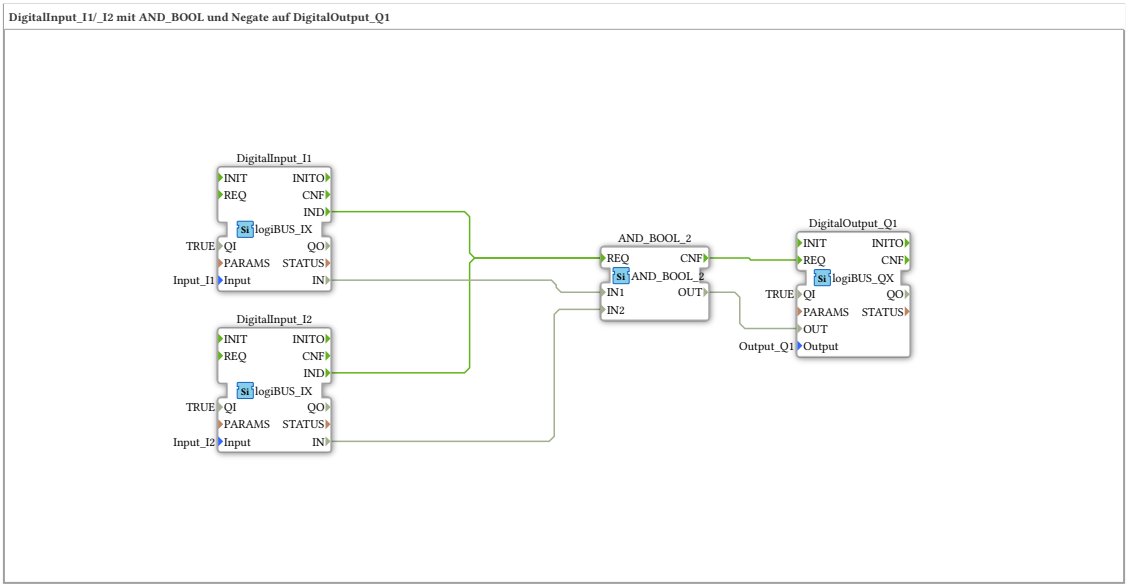
Der funktionale Ablauf:

1. Jeder Tastendruck an `I1` oder `I2` löst ein `IND`-Ereignis aus.
2. Das Ereignis triggert den `REQ`-Eingang des `AND_2_BOOL`-Bausteins.
3. Der Baustein liest die aktuellen Zustände beider Eingänge und verknüpft sie logisch (UND).
4. Nach der Berechnung sendet der Baustein ein `CNF`-Ereignis an `DigitalOutput_Q1`.
5. Der Ausgangsbaustein aktualisiert daraufhin den physischen Ausgang `Q1` mit dem berechneten Ergebnis.

Anwendungsbeispiel

Ein klassisches Anwendungsbeispiel ist eine **Zweihandbedienung zur Sicherheit**: Damit eine Maschine (`Q1`) startet, muss der Bediener zwei räumlich getrennte Taster (`I1` und `I2`) gleichzeitig drücken. Dies stellt sicher, dass beide Hände des Bedieners außerhalb des Gefahrenbereichs sind. Nur wenn beide Signale `TRUE` sind, wird der Ausgang aktiviert.

2.2.1.18 Uebung_002a4b: DigitalInput_I1/_I2 mit AND_BOOL und Negate auf DigitalOutput_Q1



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert die grundlegende Verknüpfung von digitalen Eingängen mit einem **AND**-Funktionsbaustein, sowie die Negation eines Eingangssignals. Das Ergebnis wird auf einen digitalen Ausgang geschaltet.

Ziel ist es, die Konfiguration und Verschaltung von Ein-/Ausgangsbausteinen der **logiBUS**-Familie mit einem **IEC 61131**-Standardbaustein zu verstehen.

Verwendete Funktionsbausteine (FBs)

Die Übung besteht aus folgenden direkt im Netzwerk verwendeten Funktionsbausteinen (es gibt keine Sub-Bausteine):

Baustein	Typ	Parameter
DigitalInput_I1	`logiBUS::io::DI::logiBUS_IX`	`QI = TRUE`, `Input = Input_I1`
DigitalInput_I2	`logiBUS::io::DI::logiBUS_IX`	`QI = TRUE`, `Input = Input_I2`
AND_2_BOOL	`iec61131::bitwiseOperators::AND_2_BOOL`	(keine Parameter)
DigitalOutput_Q1	`logiBUS::io::DQ::logiBUS_QX`	`QI = TRUE`, `Output = Output_Q1`

Kurzbeschreibung der verwendeten Bausteine

- **logiBUS_IX**: Liest einen digitalen Eingang der logiBUS-Hardware. Der Parameter `Input` legt den physikalischen Anschluss fest (z. B. I1, I2). Über den Ereignisausgang `IND` wird ein neuer Wert signalisiert.
- **AND_2_BOOL**: Führt eine UND-Verknüpfung auf zwei booleschen Signalen aus (Typ `iec61131::bitwiseOperators::AND_2_BOOL`). Der Ausgang `OUT` ist `TRUE` genau dann, wenn beide Eingänge `TRUE` sind.
- **logiBUS_QX**: Setzt einen digitalen Ausgang der logiBUS-Hardware. Der Parameter `Output` bestimmt den Ausgangskanal (z. B. Q1). Der Ausgang wird über das Ereignis `REQ` gesteuert.

Programmablauf und Verbindungen

Die Logik der Übung ist wie folgt aufgebaut:

1. Die digitalen Eingänge **I1** und **I2** werden über die Bausteine `DigitalInput_I1` und `DigitalInput_I2` gelesen.

2. Das Signal von **I2** wird **negiert** (invertiert). Dies geschieht über eine **Negate Connection** (Attribut `Negated = "true"`) auf der Datenverbindung zwischen `DigitalInput_I2.IN` und `AND_2_BOOL.IN2`. *(Die Negation ist nur bei booleschen Datentypen möglich.)*
3. Der Baustein `AND_2_BOOL` verknüpft das Signal von **I1** (an `IN1`) mit dem negierten Signal von **I2** (an `IN2`) mittels einer UND-Operation.
4. Das Ergebnis (`AND_2_BOOL.OUT`) wird an den Dateneingang `OUT` des Ausgangsbausteins `DigitalOutput_Q1` übergeben.
5. Die Ereignissteuerung:
   - Jeder der beiden Eingangsbausteine löst bei einem neuen Wert das Ereignis `IND` aus.
   - Beide `IND`-Ereignisse sind mit dem `REQ`-Eingang des **AND_2_BOOL** verbunden, sodass die UND-Verknüpfung bei jedem Eingangswechsel neu berechnet wird.
   - Der Ausgangsbaustein `DigitalOutput_Q1` erhält über sein `REQ`-Ereignis vom `CNF`-Ausgang des **AND_2_BOOL** den Befehl, den Ausgang zu aktualisieren.

Zusammenfassung der Logik: `Q1 = I1 AND (NOT I2)`
Lernziele:

- Konfiguration von logiBUS-Ein-/Ausgangsbausteinen.
- Verwendung des IEC-61131-Bausteins `AND_2_BOOL`.
- Anwendung einer Datennegation (Not-Verbindung) in 4diac.
- Verständnis der Ereignissteuerung mit `IND` und `REQ`.

Schwierigkeitsgrad: Einfach **Benötigte Vorkenntnisse:** Grundlagen der IEC-61499-Ereignissteuerung, einfache boolesche Logik.

Zusammenfassung

Die Übung **Uebung_002a4b** realisiert eine einfache UND-Verknüpfung mit negiertem zweiten Eingang. Sie vermittelt die Grundlagen der Verschaltung von Hardware-Bausteinen (logiBUS) mit Logikbausteinen und die Anwendung von Negationsattributen auf Datenverbindungen. Das Verhalten ist deterministisch: Der Ausgang **Q1** ist genau dann aktiv, wenn Eingang **I1** aktiv und Eingang **I2** inaktiv ist.

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

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

2.2.1.19 Uebung_002a5b: DigitalInput_I1-3 mit OR auf DigitalOutput_Q1-3



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_002a5b`. In dieser Übung werden zwei Konzepte kombiniert: Eine logische ODER-Verknüpfung (OR) mit drei Eingängen und die gleichzeitige Verteilung (Fan-Out) des Ergebnisses auf drei digitale Ausgänge.

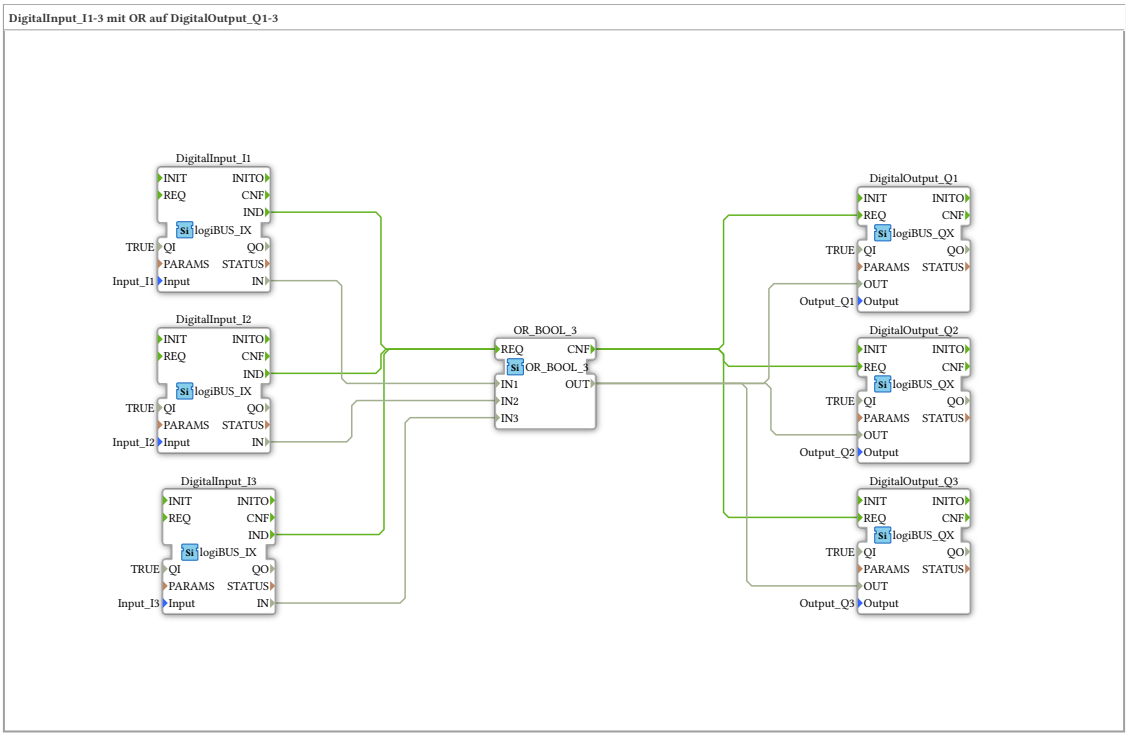
Ziel der Übung

Das Ziel ist es, eine komplexe E/A-Struktur abzubilden. Es wird gezeigt, wie Informationen von mehreren Sensoren gesammelt, logisch bewertet und das Ergebnis an eine Gruppe von Aktoren verteilt wird. Dabei wird die Skalierbarkeit von Ereignisverbindungen sowohl auf der Eingangsseite (Fan-In) als auch auf der Ausgangsseite (Fan-Out) verdeutlicht.

Beschreibung und Komponenten

In der Subapplikation `Uebung_002a5b.SUB` werden drei Eingangsbausteine über ein ODER-Gatter mit drei Ausgangsbausteinen verknüpft.

Funktionsbausteine (FBs)



- **`DigitalInput_I1` bis `I3`**: Drei Instanzen des Typs `logiBUS_IX`. Sie überwachen die Hardware-Eingänge `Input_I1`, `Input_I2` und `Input_I3`.
- **`OR_3_BOOL`**: Eine Instanz des Typs `OR_3_BOOL` (aus der IEC 61131-Bibliothek). Dieser Baustein führt eine ODER-Operation für drei boolesche Eingänge aus. Er reagiert auf `REQ` und quittiert mit `CNF`.
- **`DigitalOutput_Q1` bis `Q3`**: Drei Instanzen des Typs `logiBUS_QX`. Sie steuern die physischen Ausgänge `Output_Q1`, `Output_Q2` und `Output_Q3`.

Funktionsweise

Die Schaltung nutzt ein zentrales Logik-Element als Knotenpunkt für alle Signale. Der Aufbau in `Uebung_002a5b.SUB` ist wie folgt definiert:

```
<EventConnections>
    <Connection Source="DigitalInput_I1.IND" Destination="OR_3_BOOL.REQ"/>
    <Connection Source="DigitalInput_I2.IND" Destination="OR_3_BOOL.REQ"/>
    <Connection Source="DigitalInput_I3.IND" Destination="OR_3_BOOL.REQ"/>
    <Connection Source="OR_3_BOOL.CNF" Destination="DigitalOutput_Q1.REQ"/>
    <Connection Source="OR_3_BOOL.CNF" Destination="DigitalOutput_Q2.REQ"/>
    <Connection Source="OR_3_BOOL.CNF" Destination="DigitalOutput_Q3.REQ"/>
</EventConnections>
<DataConnections>
    <Connection Source="DigitalInput_I1.IN" Destination="OR_3_BOOL.IN1"/>
    <Connection Source="DigitalInput_I2.IN" Destination="OR_3_BOOL.IN2"/>
    <Connection Source="DigitalInput_I3.IN" Destination="OR_3_BOOL.IN3"/>
    <Connection Source="OR_3_BOOL.OUT" Destination="DigitalOutput_Q1.OUT"/>
    <Connection Source="OR_3_BOOL.OUT" Destination="DigitalOutput_Q2.OUT"/>
    <Connection Source="OR_3_BOOL.OUT" Destination="DigitalOutput_Q3.OUT"/>
</DataConnections>
```

Der funktionale Ablauf:

1. **Eingangs-Trigger**: Jede Änderung an einem der drei Taster (`I1`, `I2`, `I3`) löst eine Neuberechnung der Logik aus.
2. **Berechnung**: Der Baustein `OR_3_BOOL` setzt sein Ergebnis auf `TRUE`, wenn mindestens ein Eingang aktiv ist.
3. **Massen-Update**: Das resultierende Signal wird zeitgleich an alle drei Lampen (`Q1`, `Q2`, `Q3`) gesendet. Sobald die Logik fertig ist (`CNF`), werden alle drei Hardware-Ausgänge synchron aktualisiert.

Anwendungsbeispiel

Zentrale Warnanlage: In einer Werkshalle gibt es drei Not-Halt-Taster (`I1`, `I2`, `I3`). Sobald einer dieser Taster gedrückt wird, müssen an drei verschiedenen Stellen der Halle Warnlampen (`Q1`, `Q2`, `Q3`) aufleuchten. Die ODER-Logik sammelt die Alarme ein, und der Fan-Out sorgt für die weitreichende Signalisierung.

2.2.1.20 Uebung_002b2: DigitalInput_I1/_I2/_I3 auf DigitalOutput_Q1; mit MOVE



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_002b2`. In dieser Übung wird eine kombinatorische Logikschaltung implementiert, die zwei Grundoperationen (UND und ODER) miteinander verknüpft, wobei ein `F_MOVE`-Baustein zur expliziten Datenweiterleitung genutzt wird.

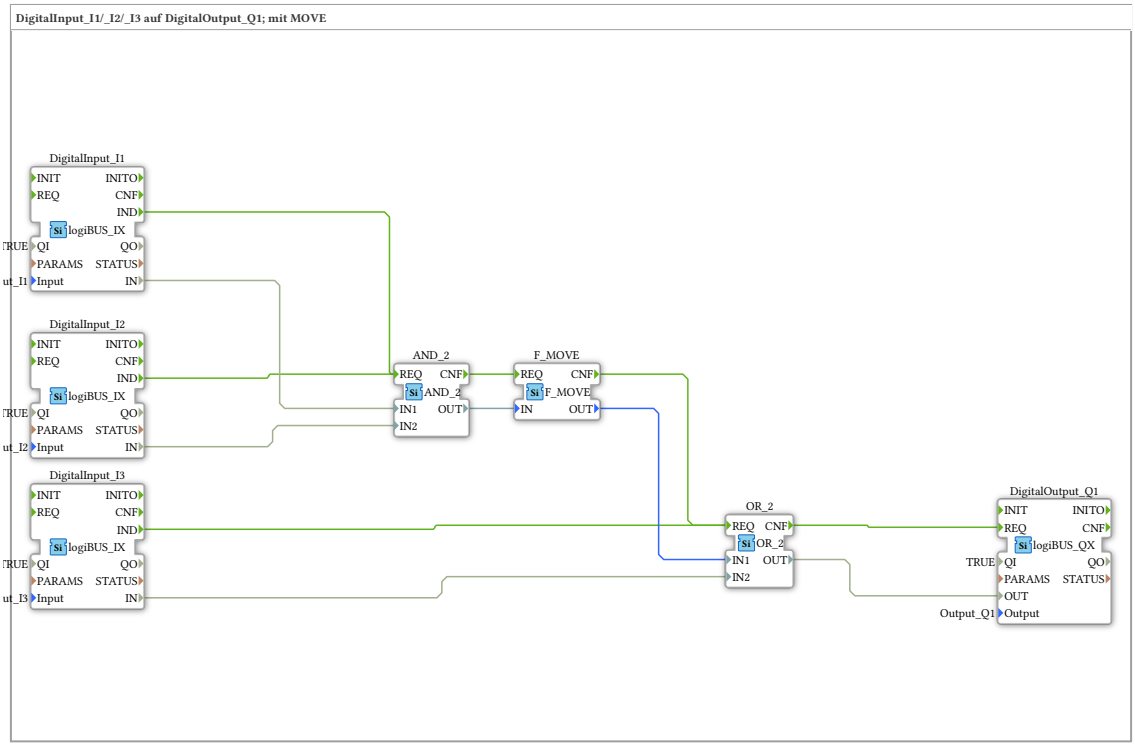
Ziel der Übung

Das Hauptziel dieser Übung ist die hierarchische Verknüpfung von Logikbausteinen. Es wird gezeigt, wie Teilergebnisse einer Operation als Eingangsgröße für eine weitere Operation dienen können. Zusätzlich wird der Baustein `F_MOVE` eingeführt, der dazu dient, Datenwerte explizit in einem eigenen Ereignisschritt weiterzureichen.

Beschreibung und Komponenten

Die Subapplikation `Uebung_002b2.SUB` realisiert die logische Funktion `Q1 = (I1 AND I2) OR I3` unter Verwendung von Standard-Logikbausteinen.

Funktionsbausteine (FBs)



- **`DigitalInput_I1 bis I3`**: Drei Instanzen des Typs `logiBUS_IX`. Sie liefern die Eingangssignale für die Logikkette.
- **`AND_2`**: Eine Instanz des Typs `AND_2`. Verknüpft die Eingänge `I1` und `I2`.
- **`F_MOVE`**: Ein Datentransfer-Baustein. Er nimmt den Wert am Eingang `IN` entgegen und gibt ihn beim Ereignis `REQ` unverändert am Ausgang `OUT` wieder aus. Er dient hier als Puffer zwischen den Logikstufen.
- **`OR_2`**: Eine Instanz des Typs `OR_2`. Verknüpft das (gepufferte) Ergebnis des UND-Bausteins mit dem dritten Eingang `I3`.
- **`DigitalOutput_Q1`**: Gibt das Endergebnis der Logik an den Hardware-Ausgang aus.

Funktionsweise

Die hierarchische Struktur der Logik wird durch die Verschaltung der Ereigniskette in `Uebung_002b2.SUB` deutlich:

```
<EventConnections>
    <Connection Source="DigitalInput_I1.IND" Destination="AND_2.REQ"/>
    <Connection Source="DigitalInput_I2.IND" Destination="AND_2.REQ"/>
    <Connection Source="AND_2.CNF" Destination="F_MOVE.REQ"/>
    <Connection Source="F_MOVE.CNF" Destination="OR_2.REQ"/>
    <Connection Source="DigitalInput_I3.IND" Destination="OR_2.REQ"/>
    <Connection Source="OR_2.CNF" Destination="DigitalOutput_Q1.REQ"/>
</EventConnections>
<DataConnections>
    <Connection Source="DigitalInput_I1.IN" Destination="AND_2.IN1"/>
    <Connection Source="DigitalInput_I2.IN" Destination="AND_2.IN2"/>
    <Connection Source="AND_2.OUT" Destination="F_MOVE.IN"/>
    <Connection Source="F_MOVE.OUT" Destination="OR_2.IN1"/>
    <Connection Source="DigitalInput_I3.IN" Destination="OR_2.IN2"/>
    <Connection Source="OR_2.OUT" Destination="DigitalOutput_Q1.OUT"/>
</DataConnections>
```

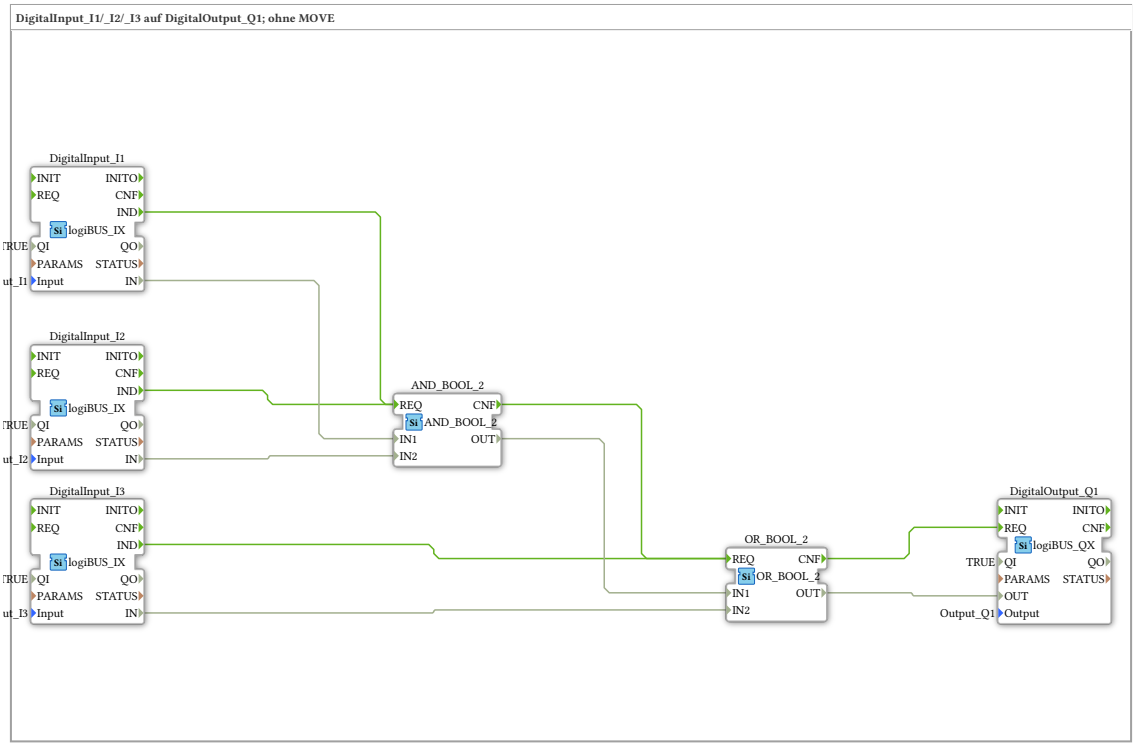
Der funktionale Ablauf:

1. Ändert sich `I1` oder `I2`, berechnet `AND_2` das Teilergebnis.
2. Das Fertigstellungs-Event (`CNF`) von `AND_2` triggert den `F_MOVE`.
3. `F_MOVE` schiebt das Teilergebnis weiter zum ODER-Baustein und triggert diesen wiederum an (`CNF -> REQ`).
4. Der ODER-Baustein verarbeitet das gepufferte Ergebnis zusammen mit dem Signal von `I3`.
5. Der Ausgang `Q1` wird aktiviert, wenn entweder beide ersten Eingänge aktiv sind ODER wenn der dritte Eingang aktiv ist.

Anwendungsbeispiel

Anlagenfreigabe mit Überbrückung: Ein Motor (`Q1`) soll normalerweise nur laufen, wenn zwei Sensoren (`I1` und `I2`) gleichzeitig grünes Licht geben (z.B. Druck ok UND Temperatur ok). Für Wartungszwecke soll der Motor jedoch auch dann gestartet werden können, wenn ein manueller Taster (`I3`) gedrückt wird, der die Automatiklogik überbrückt.

2.2.1.21 Uebung_002b3: DigitalInput_I1/_I2/_I3 auf DigitalOutput_Q1; ohne MOVE



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert die Verarbeitung digitaler Eingangssignale und deren logische Verknüpfung zur Steuerung eines digitalen Ausgangs. Die Schaltung kombiniert UND- und ODER-Verknüpfungen, um eine spezifische Logik zwischen drei Eingängen und einem Ausgang zu realisieren.

Verwendete Funktionsbausteine (FBs)

DigitalInput_I1, DigitalInput_I2, DigitalInput_I3

- **Typ**: logiBUS_IX
- **Parameter**:
  - QI = TRUE
  - Input = logiBUS_DI::Input_I1 (bzw. I2, I3)
- **Funktionsweise**: Lesen der digitalen Eingangssignale von den entsprechenden Hardware-Eingängen

AND_2_BOOL

- **Typ**: AND_2_BOOL
- **Funktionsweise**: Führt eine logische UND-Verknüpfung zwischen zwei booleschen Eingängen durch

OR_2_BOOL

- **Typ**: OR_2_BOOL
- **Funktionsweise**: Führt eine logische ODER-Verknüpfung zwischen zwei booleschen Eingängen durch

DigitalOutput_Q1

- **Typ**: logiBUS_QX
- **Parameter**:
  - QI = TRUE
  - Output = logiBUS_DO::Output_Q1
- **Funktionsweise**: Schreibt das Ergebnis der logischen Verknüpfung auf den digitalen Ausgang Q1

Programmablauf und Verbindungen

Ereignisverbindungen:

- DigitalInput_I1.IND ⟶ AND_2_BOOL.REQ
- DigitalInput_I2.IND ⟶ AND_2_BOOL.REQ
- DigitalInput_I3.IND ⟶ OR_2_BOOL.REQ
- AND_2_BOOL.CNF ⟶ OR_2_BOOL.REQ
- OR_2_BOOL.CNF ⟶ DigitalOutput_Q1.REQ

Datenverbindungen:

- DigitalInput_I1.IN ⟶ AND_2_BOOL.IN1
- DigitalInput_I2.IN ⟶ AND_2_BOOL.IN2
- DigitalInput_I3.IN ⟶ OR_2_BOOL.IN2
- AND_2_BOOL.OUT ⟶ OR_2_BOOL.IN1
- OR_2_BOOL.OUT ⟶ DigitalOutput_Q1.OUT

Logische Funktion: Q1 = (I1 UND I2) ODER I3
Lernziele:

- Verständnis der logischen Verknüpfungen UND und ODER
- Umgang mit digitalen Ein- und Ausgängen in 4diac
- Aufbau von kombinatorischen Schaltungen
- Ereignisgesteuerte Datenverarbeitung

Schwierigkeitsgrad: Einfach **Benötigte Vorkenntnisse**: Grundlagen der Digitaltechnik, Basiswissen 4diac-IDE

Start der Übung: Die Übung wird in der 4diac-IDE geladen und auf ein kompatibles Steuerungssystem deployt. Die digitalen Eingänge I1, I2 und I3 können getestet werden, um die Funktionalität der Schaltung zu überprüfen.

Zusammenfassung

Diese Übung zeigt eine grundlegende kombinatorische Logikschaltung, die digitale Eingangssignale verarbeitet und über logische Verknüpfungen einen Ausgang steuert. Die Implementierung demonstriert die prinzipielle Arbeitsweise ereignisgesteuerter Systeme in der IEC 61499 mit direkter Anbindung an Hardware-Ein- und Ausgänge.

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

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

2.2.1.22 Uebung_003: DigitalInput_I1/_I2 auf DigitalOutput_Q1/_I2 - flach



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_003`. In dieser Übung werden zwei voneinander unabhängige Signalpfade realisiert, wobei jeder digitale Eingang direkt einen zugeordneten digitalen Ausgang steuert.

Ziel der Übung

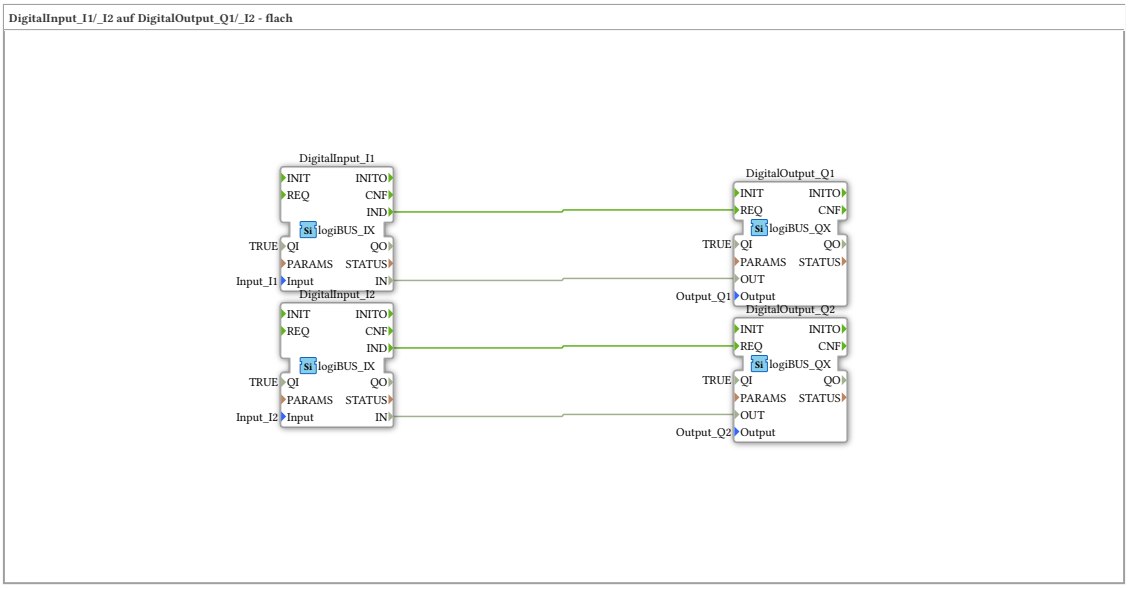
Das Hauptziel dieser Übung ist es, die parallele Verarbeitung von Signalen in der IEC 61499 zu demonstrieren. Da Funktionsbausteine in 4diac ereignisbasiert arbeiten, können mehrere Steuerungsaufgaben völlig unabhängig voneinander in einem Netzwerk existieren, ohne sich gegenseitig in der Ausführung zu blockieren.

Beschreibung und Komponenten

Die Subapplikation `Uebung_003.SUB` definiert zwei separate Signalwege („Kanäle“), die parallel verarbeitet werden.

Funktionsbausteine (FBs)

Es werden zwei Paare von Ein- und Ausgangsbausteinen verwendet:



- **`DigitalInput_I1` & `DigitalOutput_Q1`**: Das erste Paar (Kanal 1). Verbindet Hardware-Eingang `I1` mit Hardware-Ausgang `Q1`.
- **`DigitalInput_I2` & `DigitalOutput_Q2`**: Das zweite Paar (Kanal 2). Verbindet Hardware-Eingang `I2` mit Hardware-Ausgang `Q2`.

Funktionsweise

Die Unabhängigkeit der beiden Kanäle wird durch die getrennten Ereignis- und Datenverbindungen in der Subapplikation `Uebung_003.SUB` sichergestellt:

```
<EventConnections>
    <Connection Source="DigitalInput_I1.IND" Destination="DigitalOutput_Q1.REQ"/>
    <Connection Source="DigitalInput_I2.IND" Destination="DigitalOutput_Q2.REQ"/>
</EventConnections>
<DataConnections>
    <Connection Source="DigitalInput_I1.IN" Destination="DigitalOutput_Q1.OUT"/>
    <Connection Source="DigitalInput_I2.IN" Destination="DigitalOutput_Q2.OUT"/>
</DataConnections>
```

Der funktionale Ablauf:

1. Ändert sich der Zustand von `I1`, feuert der erste Baustein ein `IND`-Event, welches `Q1` zur Aktualisierung auffordert.
2. Ändert sich der Zustand von `I2`, feuert der zweite Baustein ein `IND`-Event, welches `Q2` zur Aktualisierung auffordert.

Beide Prozesse laufen asynchron ab. Eine schnelle Schaltfolge auf Kanal 1 beeinflusst die Reaktionszeit von Kanal 2 in keiner Weise.

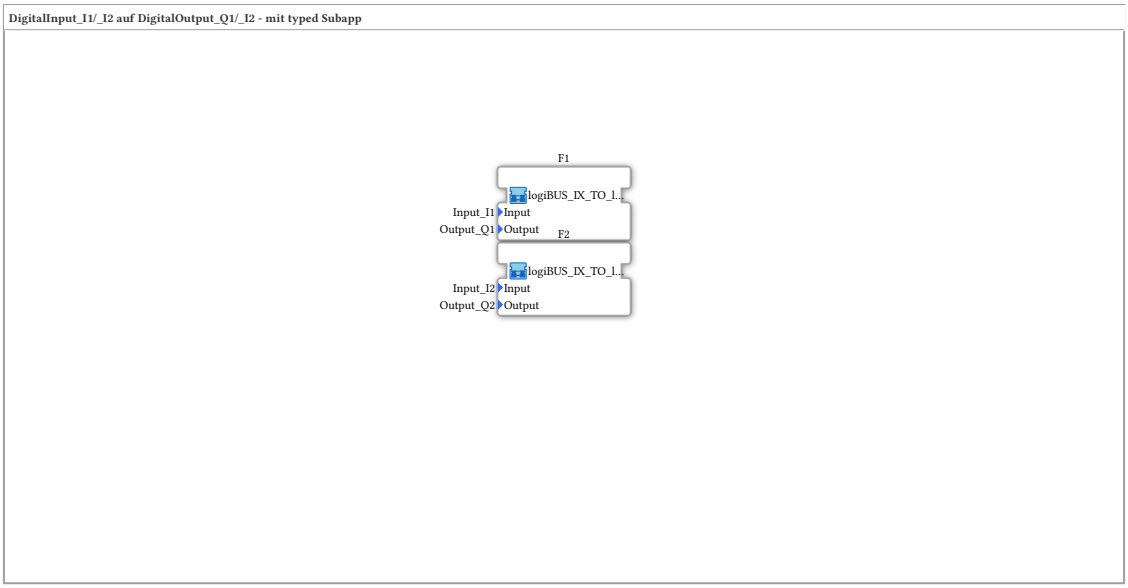
Anwendungsbeispiel

Unabhängige Aggregate: In einer landwirtschaftlichen Maschine werden zwei unabhängige Elektromotoren gesteuert. Schalter 1 (`I1`) schaltet den Motor für die Förderschnecke (`Q1`) ein, und Schalter 2 (`I2`) schaltet das Gebläse (`Q2`) ein. Obwohl beide Logiken im selben Steuerungsprogramm definiert sind, operieren sie als getrennte „Software-Schaltkreise“.

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

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

2.2.1.23 Uebung_003a: DigitalInput_I1/_I2 auf DigitalOutput_Q1/_I2 - mit typed Subapp



NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_003a`. Hier wird ein fortgeschrittener Ansatz zur Strukturierung von IEC 61499-Anwendungen demonstriert: Die Kapselung von Logik in wiederverwendbare, typisierte Sub-Applikationen („Typed SubApps“).

Ziel der Übung

Das Hauptziel ist die Vermeidung von redundantem Code. Anstatt für jeden Kanal die gleichen Bausteine und Verbindungen manuell zu zeichnen, wird einmalig ein generischer „Kanal-Typ“ definiert. Dieser kann dann beliebig oft instanziiert und individuell konfiguriert werden. Dies erhöht die Übersichtlichkeit und reduziert Fehler bei der Programmierung großer Anlagen.

Beschreibung und Komponenten

Die Subapplikation `Uebung_003a.SUB` verwendet zwei Instanzen eines selbst definierten Sub-App-Typs, um zwei Signalpfade zu realisieren.

Typisierte Sub-Applikation: `Uebung_003a_sub`

Dieser Baustein kapselt die grundlegende Logik der Signalweiterleitung. Er besitzt zwei Parameter zur Hardware-Zuordnung:

- **`Input`**: Bestimmt den physischen Eingang (z.B. `Input_I1`).
- **`Output`**: Bestimmt den physischen Ausgang (z.B. `Output_Q1`).

Im Inneren des Typs befinden sich ein `logiBUS_IX` und ein `logiBUS_QX` Baustein, die über eine Event- und eine Data-Connection fest miteinander verbunden sind.

Instanzen in der Hauptanwendung

In `Uebung_003a` werden zwei Instanzen dieses Typs platziert:

- **`F1`**: Parametriert für den Pfad `I1` zu `Q1`.
- **`F2`**: Parametriert für den Pfad `I2` zu `Q2`.

Funktionsweise

Die Komplexität der Einzelverbindungen ist im Inneren der Sub-Applikation verborgen („Information Hiding“). Die Hauptanwendung definiert lediglich die Zuweisung der physischen Adressen. Der Aufbau in `Uebung_003a.SUB` ist daher extrem kompakt:

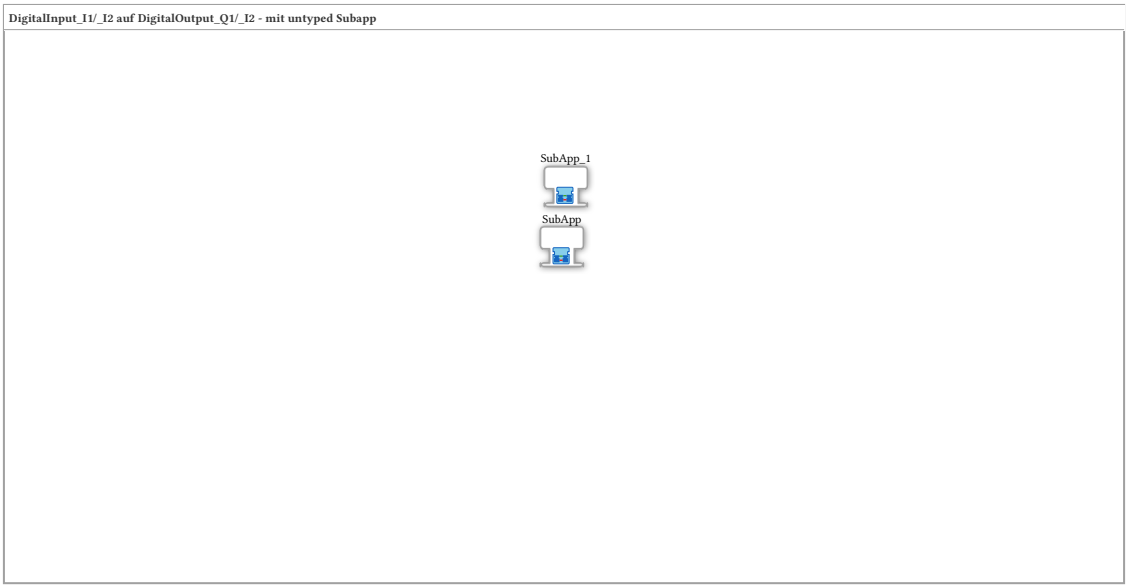
```
<SubApp Name="F1" Type="Uebungen::Uebung_003a_sub">
    <Parameter Name="Input" Value="Input_I1"/>
    <Parameter Name="Output" Value="Output_Q1"/>
</SubApp>
<SubApp Name="F2" Type="Uebungen::Uebung_003a_sub">
    <Parameter Name="Input" Value="Input_I2"/>
    <Parameter Name="Output" Value="Output_Q2"/>
</SubApp>
```

Funktional verhält sich die Anwendung exakt wie die flache Struktur in Übung 003. Jede Instanz arbeitet als eigenständiger Block, der auf Ereignisse an seinem zugewiesenen Hardware-Eingang reagiert und den Hardware-Ausgang aktualisiert.

Anwendungsbeispiel

Objektorientierte Anlagensteuerung: Stellen Sie sich eine Förderbandanlage mit 20 identischen Sektionen vor. Anstatt 20 mal die gleiche Logik zu zeichnen, erstellt man einen Typ „Sektion“. In der Hauptanwendung platziert man 20 Instanzen und gibt ihnen nur die Start-Adressen der jeweiligen Hardware-IOs. Muss später die Logik geändert werden (z.B. eine zusätzliche Zeitverzögerung), ändert man dies nur an einer einzigen Stelle (im Typ), und alle 20 Sektionen übernehmen die Änderung sofort.

2.2.1.24 Uebung_003a0: DigitalInput_I1/_I2 auf DigitalOutput_Q1/_I2 - mit untyped Subapp



NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_003a0`. Im Gegensatz zur Verwendung von Typen (wie in `Uebung_003a`) wird hier gezeigt, wie man Logik visuell gruppieren kann, ohne separate Typ-Definitionen in der Bibliothek anzulegen. Dies geschieht durch sogenannte „Untyped SubApps“.

Ziel der Übung

Das Hauptziel dieser Übung ist es, Möglichkeiten zur rein visuellen Strukturierung von Anwendungen aufzuzeigen. Untypisierte Sub-Applikationen dienen als „Container“ oder Ordner innerhalb eines Netzwerks, um zusammengehörige Funktionen zu kapseln. Sie helfen, komplexe Diagramme aufzuräumen („Aufräumen durch Einklappen“), ohne dass man sich Gedanken über Wiederverwendbarkeit oder Schnittstellen-Definitionen machen muss.

Beschreibung und Komponenten

Die Subapplikation `Uebung_003a0.SUB` enthält zwei eingebettete Sub-Applikationen (`SubApp` und `SubApp_1`), die jeweils einen eigenen Signalpfad beinhalten.

Untypisierte Sub-Applikationen

Anders als bei typisierten SubApps, die auf einer externen Definition basieren, existiert die Logik dieser Container nur innerhalb dieser spezifischen Instanz. Sie haben keine externen Ein- oder Ausgänge im Interface (in diesem Beispiel), sondern greifen direkt auf die Hardware zu.

1. **Container `SubApp`**:
   - Beinhaltet die Logik für Kanal 2.
   - Intern: `DigitalInput_I2` verbunden mit `DigitalOutput_Q2`.
2. **Container `SubApp_1`**:
   - Beinhaltet die Logik für Kanal 1.
   - Intern: `DigitalInput_I1` verbunden mit `DigitalOutput_Q1`.

Funktionsbausteine (FBs)

Innerhalb der Container werden die bekannten Standard-Bausteine verwendet:

- **`logiBUS_IX`**: Zum ereignisbasierten Einlesen der Signale.
- **`logiBUS_QX`**: Zum ereignisbasierten Ausgeben der Signale.

Funktionsweise

Die Logik ist identisch zur parallelen Steuerung, jedoch ist die Ansicht hierarchisch gegliedert. Der Aufbau in `Uebung_003a0.SUB` zeigt die Verschachtelung:

```
<SubAppNetwork>
    <SubApp Name="SubApp">
        <SubAppNetwork>
            <FB Name="DigitalInput_I2" ... />
            <FB Name="DigitalOutput_Q2" ... />
            <EventConnections> ... </EventConnections>
            <DataConnections> ... </DataConnections>
        </SubAppNetwork>
    </SubApp>

    <SubApp Name="SubApp_1">
        <SubAppNetwork>
            <FB Name="DigitalInput_I1" ... />
            <FB Name="DigitalOutput_Q1" ... />
            <EventConnections> ... </EventConnections>
            <DataConnections> ... </DataConnections>
        </SubAppNetwork>
    </SubApp>
</SubAppNetwork>
```

Der funktionale Ablauf: Die Kapselung hat keinen Einfluss auf die Laufzeit-Ausführung. Die Bausteine verhalten sich exakt so, als wären sie alle auf der obersten Ebene platziert.

1. `SubApp_1` verarbeitet das Signal von `I1` zu `Q1`.
2. `SubApp` verarbeitet das Signal von `I2` zu `Q2`.

Anwendungsbeispiel

Visuelle Gruppierung von Funktionsbereichen: In einer großen Anlage könnte man untypisierte SubApps nutzen, um die Logik thematisch zu sortieren, z.B. einen Container „Temperaturregelung“, einen „Antriebssteuerung“ und einen „Sicherheitsüberwachung“. Wenn man die Hauptansicht öffnet, sieht man nur diese drei Blöcke und nicht hunderte von einzelnen Gattern und Verbindungen. Dies erleichtert die Navigation („Zoom-In“-Effekt), ist aber nicht für die Wiederverwendung über mehrere Projekte hinweg gedacht (dafür nutzt man typisierte SubApps).

2.2.1.25 Uebung_003a_sub

Hinweis

Diese Übung existiert in den aktuellen Trainingsquellen nicht mehr (entfernt, zusammengelegt oder umbenannt). Diese Seite bleibt nur als Platzhalter für bestehende Verweise erhalten. Die aktuellen Übungen stehen in der Übungsübersicht.

2.2.1.26 Uebung_003b: DigitalInput_I1-2-3-4 auf DigitalOutput_Q1-2-3-4



NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_003b`. Diese Übung erweitert das Konzept der modularen Steuerung aus `Uebung_003a` auf insgesamt vier unabhängige Kanäle.

Ziel der Übung

Das Hauptziel ist die Demonstration der Skalierbarkeit durch Typisierung. Es wird gezeigt, wie einfach eine bestehende Logik vervielfältigt werden kann, indem ein bereits definierter Sub-App-Typ mehrfach instanziiert wird. Dies unterstreicht den Effizienzvorteil der IEC 61499 bei der Verwaltung vieler gleichartiger Hardware-Schnittstellen.

Beschreibung und Komponenten

Die Subapplikation `Uebung_003b.SUB` verwendet vier Instanzen des Typs `Uebung_003b_sub` (identisch zu `Uebung_003a_sub`), um vier I/O-Paare zu verknüpfen.

Typisierte Sub-Applikation: `Uebung_003b_sub`

Dieser Baustein kapselt die 1:1 Weiterleitung von einem digitalen Eingang (`IX`) zu einem digitalen Ausgang (`QX`). Er stellt zwei Parameter für die Konfiguration bereit:

- **`Input`**: Auswahl des physischen Eingangs (z.B. `Input_I1` bis `Input_I4`).
- **`Output`**: Auswahl des physischen Ausgangs (z.B. `Output_Q1` bis `Output_Q4`).

Instanzen in der Hauptanwendung

In `Uebung_003b` werden vier Instanzen (`F1` bis `F4`) genutzt:

- **`F1`**: `I1` ➡️ `Q1`
- **`F2`**: `I2` ➡️ `Q2`
- **`F3`**: `I3` ➡️ `Q3`
- **`F4`**: `I4` ➡️ `Q4`

Funktionsweise

Die Logik ist vollständig in den Sub-Applikationen gekapselt. Die Hauptanwendung dient nur noch als Konfigurations-Ebene. Der Aufbau in `Uebung_003b.SUB` zeigt die Zuweisung der Parameter:

```
<SubApp Name="F1" Type="Uebungen::Uebung_003b_sub">
    <Parameter Name="Input" Value="Input_I1"/>
```

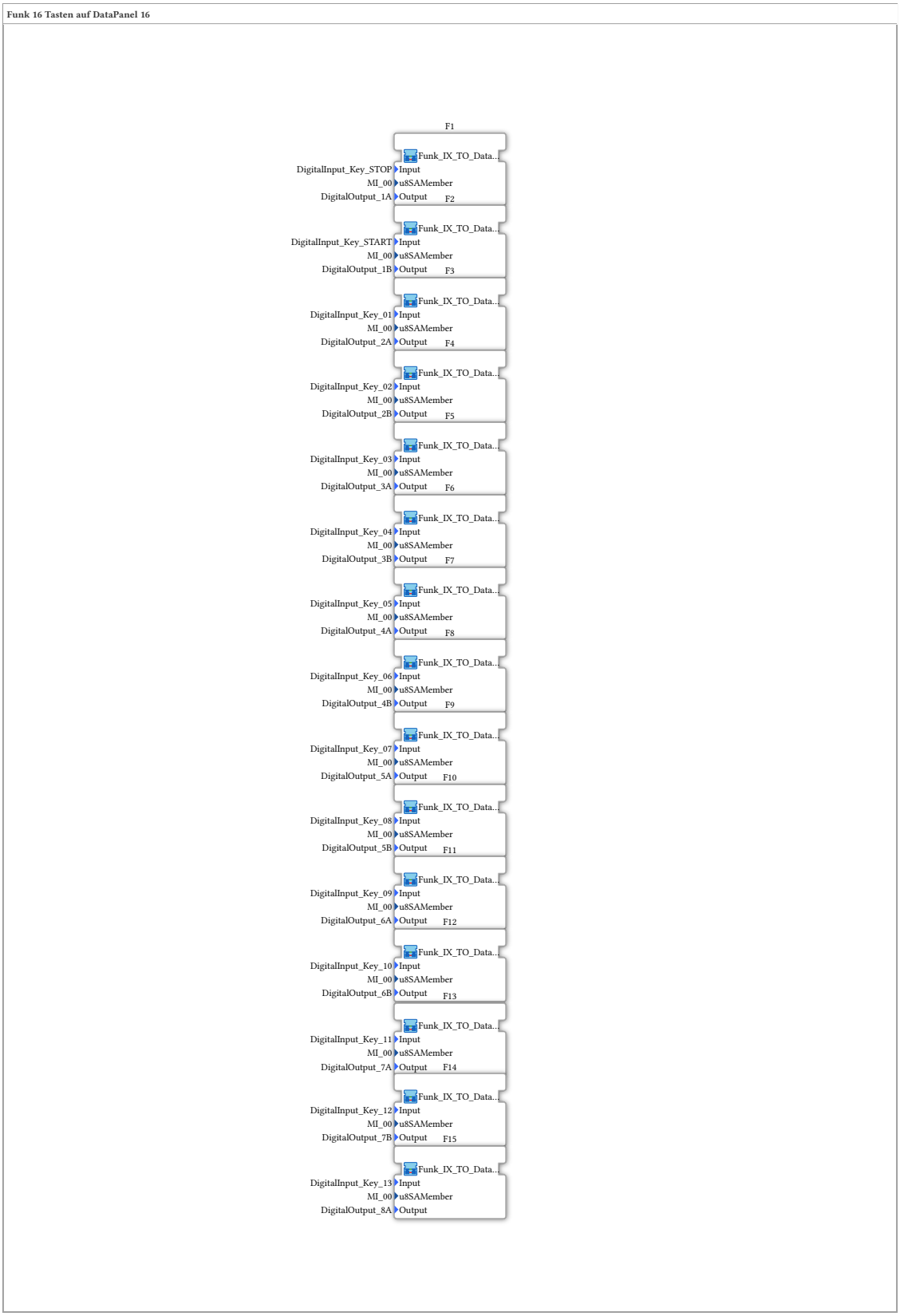
```
    <Parameter Name="Output" Value="Output_Q1"/>
</SubApp>
<!-- ... Instanzen F2 und F3 ... -->
<SubApp Name="F4" Type="Uebungen::Uebung_003b_sub">
    <Parameter Name="Input" Value="Input_I4"/>
    <Parameter Name="Output" Value="Output_Q4"/>
</SubApp>
```

Jeder Kanal arbeitet völlig autark. Wenn an Eingang `I3` eine Änderung erkannt wird, reagiert ausschließlich die Instanz `F3` und schaltet den Ausgang `Q3`. Die anderen Kanäle bleiben davon unberührt.

Anwendungsbeispiel

Modularer Schaltschrankbau: Stellen Sie sich ein IO-Modul mit 8 oder 16 Kanälen vor. Anstatt für jeden Kanal einzeln Verbindungen zu ziehen, nutzt man die typisierte Sub-Applikation als „Kanal-Treiber“. Die Software-Struktur spiegelt so exakt den modularen Aufbau der Hardware wider, was die Fehlersuche und Dokumentation massiv vereinfacht.

2.2.1.27 Uebung_003b2: Funk 16 Tasten auf DataPanel 16



NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_003b2`. In dieser Übung wird die Skalierbarkeit des Systems demonstriert, indem 15 Tasten einer Funkfernbedienung auf 15 digitale Ausgänge eines DataPanels gemappt werden.

Ziel der Übung

Das Ziel dieser Übung ist die effiziente Verwaltung einer großen Anzahl von Hardware-Schnittstellen. Durch den Einsatz von typisierten Sub-Applikationen mit exponierten Parametern kann die komplette Zuordnung (Mapping) zwischen Funk-Eingängen und CAN-Bus-Ausgängen zentral in der Hauptanwendung vorgenommen werden, ohne die interne Logik verändern zu müssen.

Beschreibung und Komponenten

Die Subapplikation `Uebung_003b2.SUB` instanziiert 15 mal den spezialisierten Sub-Typ `Uebung_003b2_sub` (benannt `F1` bis `F15`).

Typisierte Sub-Applikation: `Uebung_003b2_sub`

Dieser Baustein dient als universeller „Kanal-Treiber“ für die Funk-zu-CAN-Kommunikation. Er verfügt über folgende Parameter:

- **`Input`**: Der Name der Funk-Taste (z.B. `Key_01`, `START`, `STOP`).
- **`u8SAMember`**: Die CAN-Adresse des DataPanels (hier fest auf `MI_00` gesetzt).
- **`Output`**: Der Name des physischen Ausgangs am DataPanel (z.B. `DigitalOutput_1A`).

Intern enthält der Baustein einen `Funk_IX` zum Empfangen der Funksignale und einen `DataPanel_MI_QX` zum Senden der CAN-Nachrichten.

Konfiguration der Kanäle

In `Uebung_003b2` sind die Zuweisungen klar definiert:

- `F1`: `STOP` ➡️ `Output_1A`
- `F2`: `START` ➡️ `Output_1B`
- `F3` bis `F15`: `Key_01` bis `Key_13` ➡️ `Output_2A` bis `Output_8A`

Funktionsweise

Der Signalweg verläuft ereignisbasiert über die CAN-Bus-Infrastruktur:

1. Der Nutzer drückt eine Taste auf der Funkfernbedienung.
2. Die Instanz des Typs `Funk_IX` im Inneren der entsprechenden Sub-Applikation erkennt den Tastendruck und feuert ein `IND`-Event.
3. Dieses Ereignis wird direkt an den `REQ`-Eingang des `DataPanel_MI_QX` Bausteins geleitet.
4. Der Ausgangs-Baustein generiert daraufhin eine CAN-Nachricht für das DataPanel, um den zugeordneten physischen Ausgang ein- oder auszuschalten.

Da alle 15 Instanzen parallel und unabhängig arbeiten, können beliebig viele Tasten gleichzeitig bedient werden.

Anwendungsbeispiel

Nachrüstung einer Funkfernsteuerung: Eine bestehende Maschine soll mit einer Funkfernbedienung für 15 verschiedene Hydraulikfunktionen ausgestattet werden. Anstatt die gesamte Logik neu zu programmieren, nutzt man das DataPanel als IO-Knoten und mappt die Funktasten rein parametrisch auf die Ventilausgänge. Dies ermöglicht eine extrem schnelle Inbetriebnahme und einfache Anpassung an Kundenwünsche (z.B. Tastenbelegung ändern).

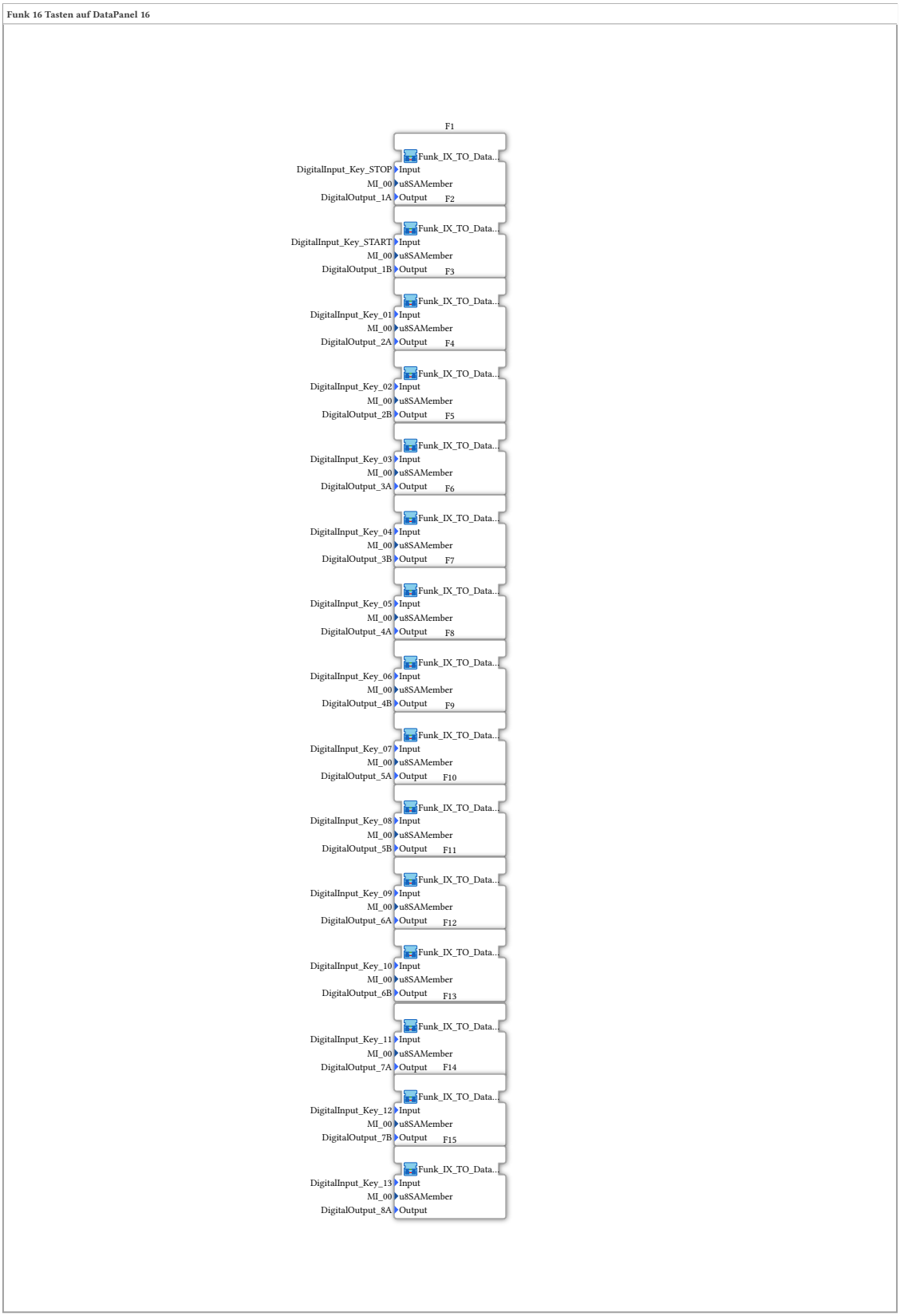
2.2.1.28 Uebung_003b2_sub

Hinweis

Diese Übung existiert nicht mehr als eigene Übung. Der Baustein wurde in die Bibliothek verschoben und heißt jetzt Funk_IX_TO_DataPanel_MI_QX. Beschreibung, Schnittstelle und Netzwerk stehen dort.

2.2.1.29 Uebung_003b3: Funk 16 Tasten auf DataPanel 16

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_003b3 (Funk 16 Tasten auf DataPanel 16).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Funk 16 Tasten auf DataPanel 16**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_003b3.SUB, welche die folgende Bausteinstruktur verwendet:

Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Signalweiterleitung**: Die Bausteine verarbeiten Daten- und Steuersignale gemäß ihrer Konfiguration.

Zusammenfassung

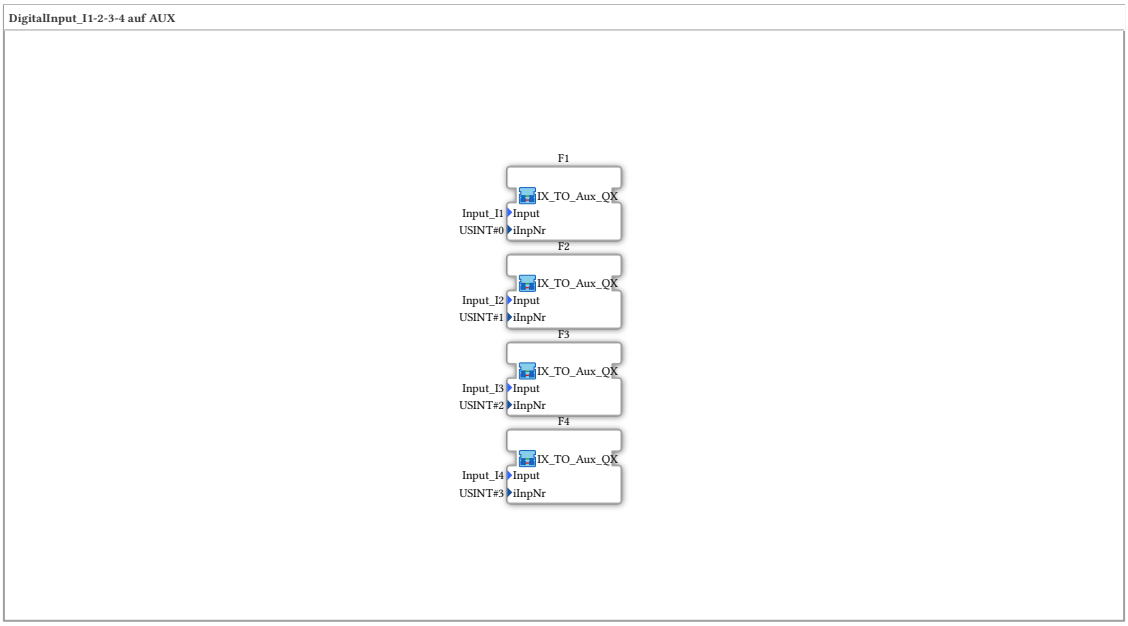
Die Übung Uebung_003b3 demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.2.1.30 Uebung_003b_sub

Hinweis

Diese Übung existiert in den aktuellen Trainingsquellen nicht mehr (entfernt, zusammengelegt oder umbenannt). Diese Seite bleibt nur als Platzhalter für bestehende Verweise erhalten. Die aktuellen Übungen stehen in der Übungsübersicht.

2.2.1.31 Uebung_003c: DigitalInput_I1-2-3-4 auf AUX



NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_003c`. Hier wird demonstriert, wie lokale Hardware-Eingänge an das ISOBUS-System als „Auxiliary Inputs“ angebunden werden, wobei eine typisierte Sub-Applikation zur Strukturierung verwendet wird.

🎧 Podcast

- Altbayerisch für Einsteiger: Von Gratler-Schnupfen und Stadthodern – Eine Laute-Reise durch Lektion 3C

Ziel der Übung

Das Ziel ist es, lokale physische Schalter (`I1` bis `I4`) für das ISOBUS-Netzwerk verfügbar zu machen. In der ISOBUS-Welt können diese Eingänge als „Auxiliary Inputs“ (Hilfseingänge) definiert werden. Der Endbenutzer kann diese dann am Terminal flexibel auf verschiedene Maschinenfunktionen mappen (z.B. „Taster 1 steuert Klappe auf/zu“).

Beschreibung und Komponenten

Die Subapplikation `Uebung_003c.SUB` nutzt vier Instanzen des Typs `Uebung_003c_sub`, um vier Kanäle für das ISOBUS-Mapping bereitzustellen.

Typisierte Sub-Applikation: `Uebung_003c_sub`

Dieser Baustein verbindet einen Standard-Digitaleingang mit einem ISOBUS-Auxiliary-Ausgang. Er verfügt über folgende Parameter:

- **`Input`**: Auswahl des lokalen Hardware-Pins (z.B. `Input_I1`).
- **`iInpNr`**: Die fortlaufende Nummer des AUX-Eingangs im Objekt-Pool (Index 0 bis n).

Intern werden die Ereignisse (`IND -> REQ`) und Daten (`IN -> OUT`) vom `logiBUS_IX` zum `Aux_QX` Baustein durchgereicht.

Konfiguration der Kanäle

In `Uebung_003c` erfolgt das Mapping wie folgt:

- `F1`: `Input_I1` ➡️ AUX Index 0
- `F2`: `Input_I2` ➡️ AUX Index 1
- `F3`: `Input_I3` ➡️ AUX Index 2
- `F4`: `Input_I4` ➡️ AUX Index 3

Funktionsweise

1. Der Bediener betätigt einen der physischen Schalter (z.B. `I1`).
2. Die entsprechende Instanz (z.B. `F1`) erkennt die Pegeländerung.
3. Ein `IND`-Event wird ausgelöst und triggert den `REQ`-Port des `Aux_QX` Bausteins.
4. Der `Aux_QX` Baustein sendet eine standardisierte ISOBUS-Nachricht in das CAN-Netzwerk, die den Status des „Auxiliary Input N“ mitteilt.
5. Das verbundene ISOBUS-Anbaugerät empfängt diese Nachricht und führt die vom Nutzer zugewiesene Aktion aus.

Anwendungsbeispiel

Nachrüstung von Bedienelementen: Ein Traktor verfügt über keine originalen ISOBUS-Joystick-Tasten. Man installiert eine kleine Konsole mit vier Standard-Tastern in der Kabine und verbindet diese mit der logiBUS-Steuerung. Dank dieser Software-Logik erscheinen die Taster für alle ISOBUS-Geräte (z.B. Feldspritze, Düngerstreuer) als vollwertige, frei belegbare Bedienelemente auf dem Terminal.

2.2.1.32 Uebung_003c_sub

Hinweis

Diese Übung existiert nicht mehr als eigene Übung. Der Baustein wurde in die Bibliothek verschoben und heißt jetzt IX_TO_Aux_QX. Beschreibung, Schnittstelle und Netzwerk stehen dort.

2.2.1.33 Uebung_003d: DigitalInput_I1/_I2 auf DigitalOutput_Q1/_I2



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_003d`. Diese Übung ist strukturell identisch mit `Uebung_003` und dient der Festigung des Verständnisses für parallele Signalpfade in der IEC 61499.

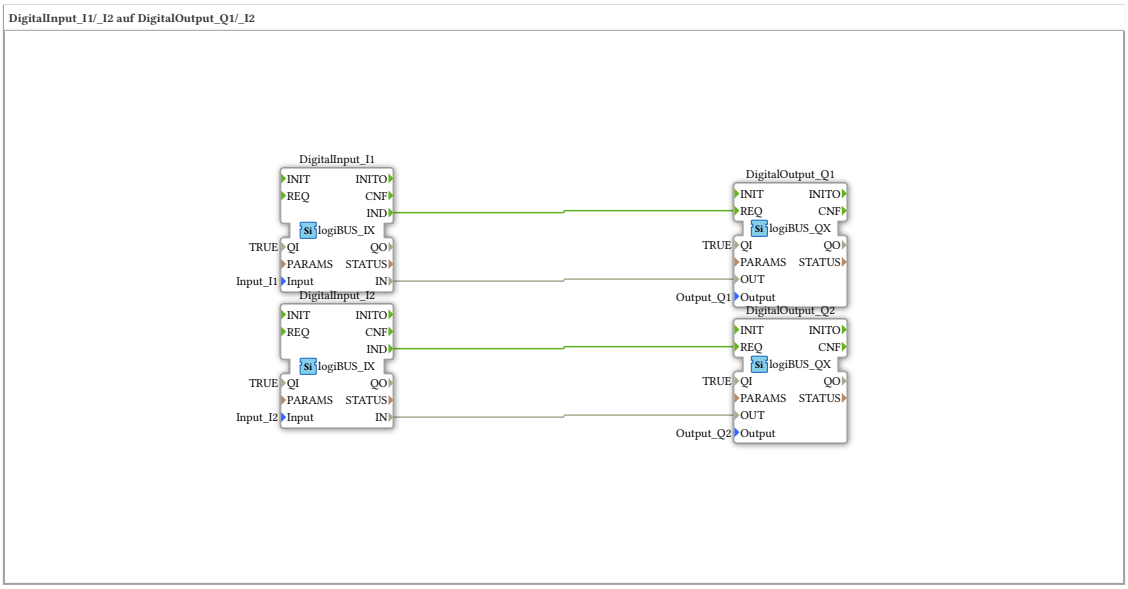
Ziel der Übung

Das Ziel ist die Wiederholung der direkten I/O-Verknüpfung mittels Ereignis- und Datenverbindungen. Es wird sichergestellt, dass das Konzept der asynchronen und unabhängigen Datenflüsse verstanden wurde.

Beschreibung und Komponenten

Die Subapplikation `Uebung_003d.SUB` verbindet zwei Eingangsbausteine direkt mit zwei Ausgangsbausteinen.

Funktionsbausteine (FBs)



- **`DigitalInput_I1` ➡️ `DigitalOutput_Q1`**
- **`DigitalInput_I2` ➡️ `DigitalOutput_Q2`**

Die Bausteintypen sind `logiBUS_IX` (Eingang) und `logiBUS_QX` (Ausgang).

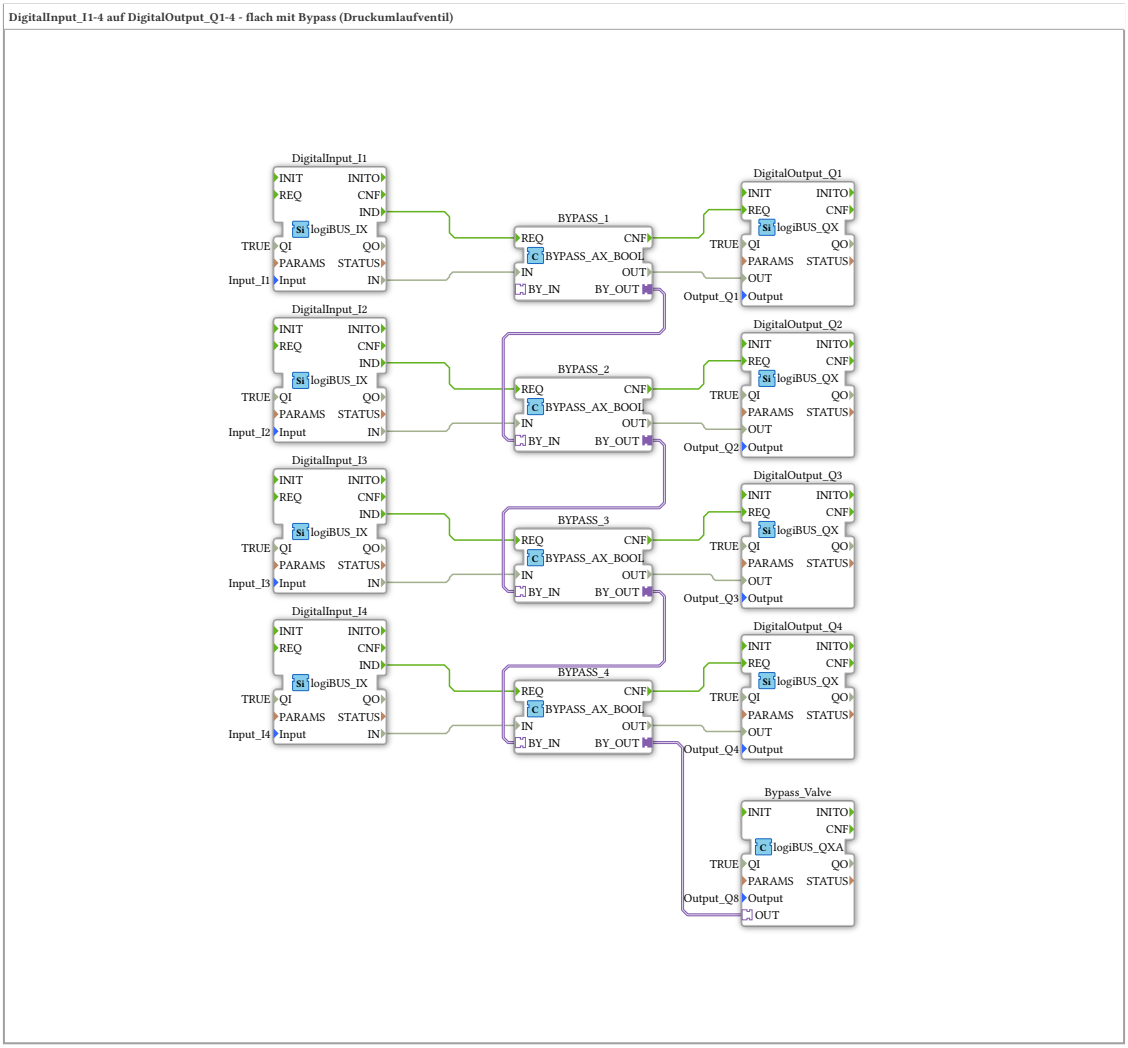
Funktionsweise

Die Signale werden 1:1 und latenzarm von den Eingängen auf die Ausgänge durchgeschleift. Jede Änderung am Eingang `I1` löst sofort eine Aktualisierung von `Q1` aus, ohne dass die Logik für `I2`/`Q2` davon beeinflusst wird.

Anwendungsbeispiel

Diese Übung eignet sich hervorragend als **Verdrahtungstest-Programm**: Wenn eine neue Hardware-Konfiguration aufgebaut wurde, lädt man dieses „transparente“ Programm hoch, um zu prüfen, ob alle Taster und Lampen physikalisch korrekt angeschlossen und adressiert sind. Ein Druck auf den Taster muss die zugehörige Lampe unmittelbar zum Leuchten bringen.

2.2.1.34 Uebung_003e: DigitalInput_I1-4 auf DigitalOutput_Q1-4 - flach mit Bypass (Druckumlaufventil)



• – ○ • – ○ • – ○ •

Einleitung

In dieser Übung werden vier digitale Eingangssignale (I1–I4) direkt auf vier digitale Ausgänge (Q1–Q4) durchgeschaltet. Dazwischen liegt jeweils ein Bypass-Baustein, der eine zusätzliche Funktionalität bietet: Die Bypass-Blöcke sind kaskadenartig miteinander verbunden und führen über ein gemeinsames Bypass-Ventil (Ausgang Q8). Die Schaltung realisiert eine einfache Durchschleifung mit der Möglichkeit, den Signalfluss durch ein Druckumlaufventil zu beeinflussen.

Verwendete Funktionsbausteine (FBs)

Die Übung besteht aus folgenden FB-Instanzen:

- **DigitalInput_I1** bis **DigitalInput_I4**
  Typ: `logiBUS::io::DI::logiBUS_IX`
  Parameter: QI = TRUE, Input = zugehöriger physikalischer Eingang (Input_I1 usw.)
- **DigitalOutput_Q1** bis **DigitalOutput_Q4**
  Typ: `logiBUS::io::DQ::logiBUS_QX`
  Parameter: QI = TRUE, Output = zugehöriger physikalischer Ausgang (Output_Q1 usw.)
- **BYPASS_1** bis **BYPASS_4**
  Typ: `logiBUS::signalprocessing::bypass::BYPASS_AX_BOOL`
  Keine Parameter gesetzt.

- **Bypass_Valve**
  Typ: `logiBUS::io::DQ::logiBUS_QXA`
  Parameter: QI = TRUE, Output = Output_Q8 (Bypass-Ventil)

Ereignisverbindungen:

- `DigitalInput_I1.IND` ⟶ `BYPASS_1.REQ`
- `BYPASS_1.CNF` ⟶ `DigitalOutput_Q1.REQ`
- (Analog für I2–I4)

Datenverbindungen:

- `DigitalInput_I1.IN` ⟶ `BYPASS_1.IN`
- `BYPASS_1.OUT` ⟶ `DigitalOutput_Q1.OUT`
- (Analog für I2–I4)

Adapterverbindungen (Bypass-Kette):

- `BYPASS_1.BY_OUT` ⟶ `BYPASS_2.BY_IN`
- `BYPASS_2.BY_OUT` ⟶ `BYPASS_3.BY_IN`
- `BYPASS_3.BY_OUT` ⟶ `BYPASS_4.BY_IN`
- `BYPASS_4.BY_OUT` ⟶ `Bypass_Valve.OUT`

Programmablauf und Verbindungen

1. **Signalweg**: Jeder Digitaleingang (I1–I4) erzeugt bei einer Änderung ein Ereignis `IND`. Dieses triggert den zugehörigen Bypass-Baustein (`BYPASS_x`). Der Bypass-Baustein gibt das Signal unverändert (oder mit Bypass-Funktion) an den Ausgang weiter und signalisiert mit `CNF` den entsprechenden Digitalausgang (`DigitalOutput_Qx`), der das physikalische Ausgangssignal setzt.
2. **Bypass-Kette**: Die Adapteranschlüsse `BY_OUT` und `BY_IN` der vier Bypass-Blöcke sind in Reihe geschaltet. Dadurch wird ein gemeinsames Bypass-Signal (z. B. ein Freigabe- oder Sperrsignal) durch die gesamte Kette geleitet. Das letzte Glied der Kette (`BYPASS_4.BY_OUT`) ist mit dem Bypass-Ventil (`Bypass_Valve`) verbunden, das auf den Ausgang `Output_Q8` wirkt.
3. **Funktion des Bypass**: Durch das Bypass-Signal kann der gesamte Datenfluss der vier Kanäle zentral beeinflusst werden – beispielsweise ein- oder ausgeschaltet. In der Konfiguration als Druckumlaufventil dient dies zur Steuerung eines hydraulischen oder pneumatischen Kreislaufs.

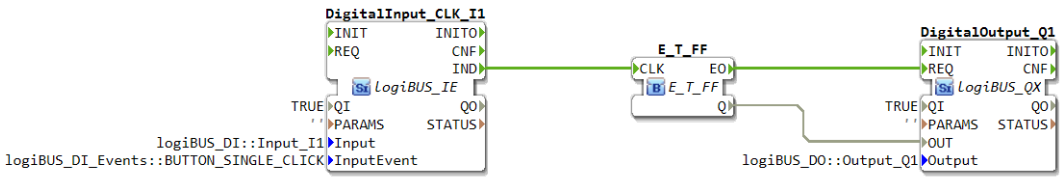
Zusammenfassung

Die Übung zeigt die grundlegende Verkettung von digitalen Ein- und Ausgängen mit Zwischenschaltung von Bypass-Bausteinen. Die Besonderheit liegt in der kaskadierten Adapterverbindung, die es erlaubt, ein gemeinsames Steuersignal über mehrere Kanäle zu führen. Dieses Konzept eignet sich für Anwendungen, bei denen eine zentrale Abschaltung oder Umlenkung der Signale erforderlich ist, beispielsweise in Sicherheitsschaltungen oder Druckregelungen.

2.2.1.35 Uebung_004a: Toggle Flip-Flop mit IE mit BUTTON_SINGLE_CLICK

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_004a`. In dieser Übung verlassen wir die reine Datenweiterleitung und nutzen Ereignisse (Events), um eine Speicherfunktion zu realisieren: Einen klassischen Stromstoßschalter.



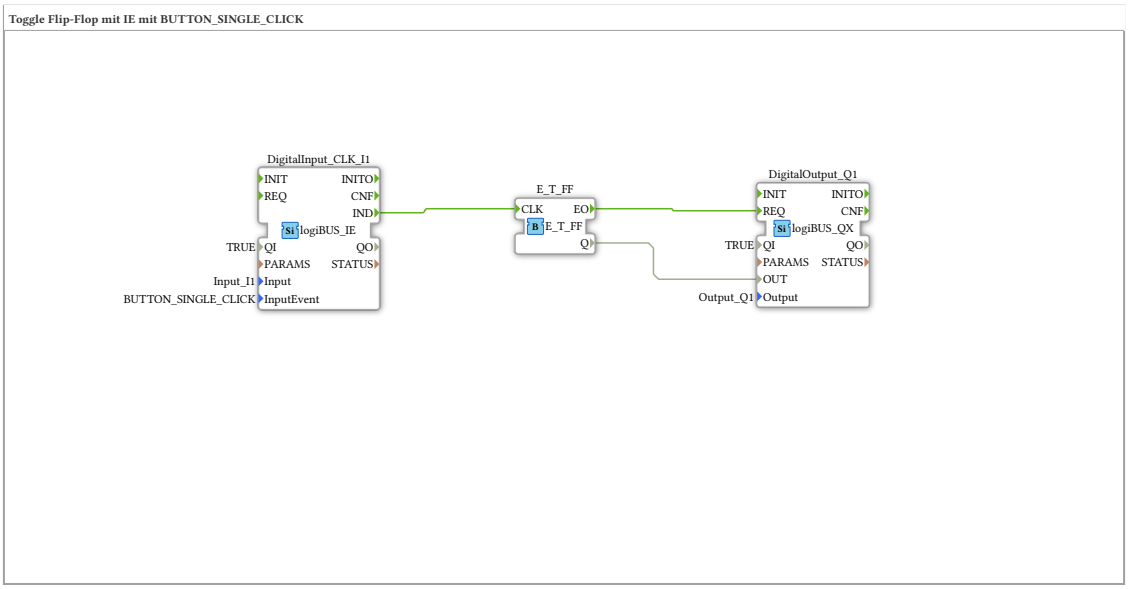
Ziel der Übung

Das Ziel ist es, den Unterschied zwischen zustandsorientierter (Pegel) und ereignisorientierter (Flanke) Programmierung zu verstehen. Während ein einfacher Taster nur solange „Ein“ ist, wie er gedrückt wird, soll hier jeder Tastendruck den Zustand des Ausgangs wechseln (Umschalten: Aus ➡️ Ein ➡️ Aus ➡️ ...).

Beschreibung und Komponenten

Die Subapplikation `Uebung_004a.SUB` verwendet einen speziellen Eingangsbaustein, der Klick-Ereignisse generiert, und ein Toggle-Flip-Flop.

Funktionsbausteine (FBs)



- **`DigitalInput_CLK_I1`**: Typ `logiBUS_IE` (Input Event). Im Gegensatz zum Standard-Eingang liefert dieser Baustein kein kontinuierliches Signal, sondern feuert ein einzelnes Ereignis (`IND`), wenn eine bestimmte Bedingung erfüllt ist. Hier ist er auf `BUTTON_SINGLE_CLICK` konfiguriert.
- **`E_T_FF`**: Typ `E_T_FF` (Standard-IEC-Event-Baustein). Dieser Baustein hat einen Takteingang (`CLK`). Bei jedem empfangenen Ereignis wechselt er seinen internen Zustand und gibt diesen über den Daten-Ausgang `Q` sowie ein Bestätigungs-Event `EO` aus.
- **`DigitalOutput_Q1`**: Typ `logiBUS_QX`. Schaltet den physischen Ausgang `Q1` basierend auf dem Zustand des Flip-Flops.

Funktionsweise

Die Logik basiert auf der Umwandlung eines flüchtigen Tastendrucks in einen dauerhaften Speicherzustand:

```
<EventConnections>
    <Connection Source="DigitalInput_CLK_I1.IND" Destination="E_T_FF.CLK"/>
    <Connection Source="E_T_FF.EO" Destination="DigitalOutput_Q1.REQ"/>
</EventConnections>
<DataConnections>
    <Connection Source="E_T_FF.Q" Destination="DigitalOutput_Q1.OUT"/>
</DataConnections>
```

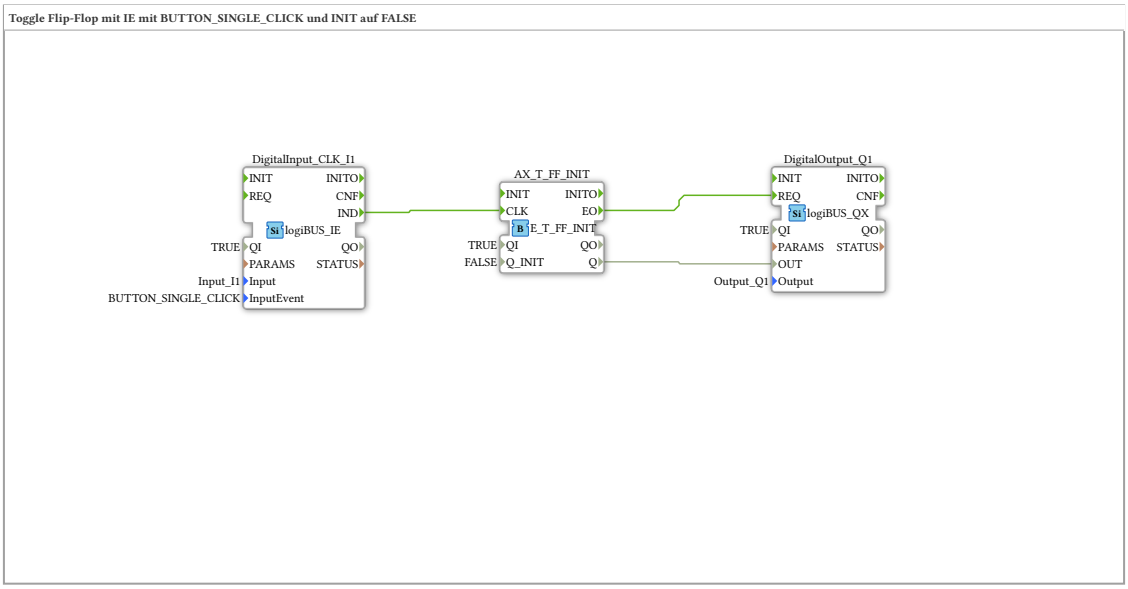
1. Der Benutzer drückt den Taster an `I1` kurz („Klick“).
2. Der `DigitalInput_CLK_I1` erkennt das Muster „Einzelklick“ und sendet ein `IND`-Ereignis.
3. Das Ereignis erreicht den `CLK`-Eingang des `E_T_FF`.
4. Das Flip-Flop kippt seinen Zustand (z.B. von FALSE auf TRUE).
5. Das neue Signal steht am Daten-Ausgang `Q` bereit und das Flip-Flop sendet ein Ereignis an `EO`.
6. `DigitalOutput_Q1` empfängt dieses Ereignis, liest den Wert von `Q` und schaltet die Lampe ein.
7. Beim nächsten Klick wiederholt sich der Vorgang, das Flip-Flop kippt zurück auf FALSE, die Lampe geht aus.

Anwendungsbeispiel

Die klassische **Flurbeleuchtung**: Ein Tasterdruck schaltet das Licht ein, der nächste schaltet es wieder aus. Dies ist mit einem rein elektrischen Taster (der zurückfedert) nicht ohne Speicherelement (Software-Flip-Flop) möglich.

2.2.1.36 Uebung_004a10a: Toggle Flip-Flop mit IE mit BUTTON_SINGLE_CLICK und INIT auf FALSE

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_004a10a (Toggle Flip-Flop mit IE mit BUTTON_SINGLE_CLICK und INIT auf FALSE).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Toggle Flip-Flop mit IE mit BUTTON_SINGLE_CLICK und INIT auf FALSE**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_004a10a.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QX.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
- **DigitalInput_CLK_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **AX_T_FF_INIT**: Instanz des Typs iec61499::events::E_T_FF_INIT.
  - Parameter QI = TRUE
  - Parameter Q_INIT = FALSE

Verbindungen und Schnittstellen

Ereignisverbindungen:

- DigitalInput_CLK_I1.IND -> AX_T_FF_INIT.CLK
- AX_T_FF_INIT.EO -> DigitalOutput_Q1.REQ

Datenverbindungen:

- AX_T_FF_INIT.Q -> DigitalOutput_Q1.OUT

Programmablauf und Funktionsweise

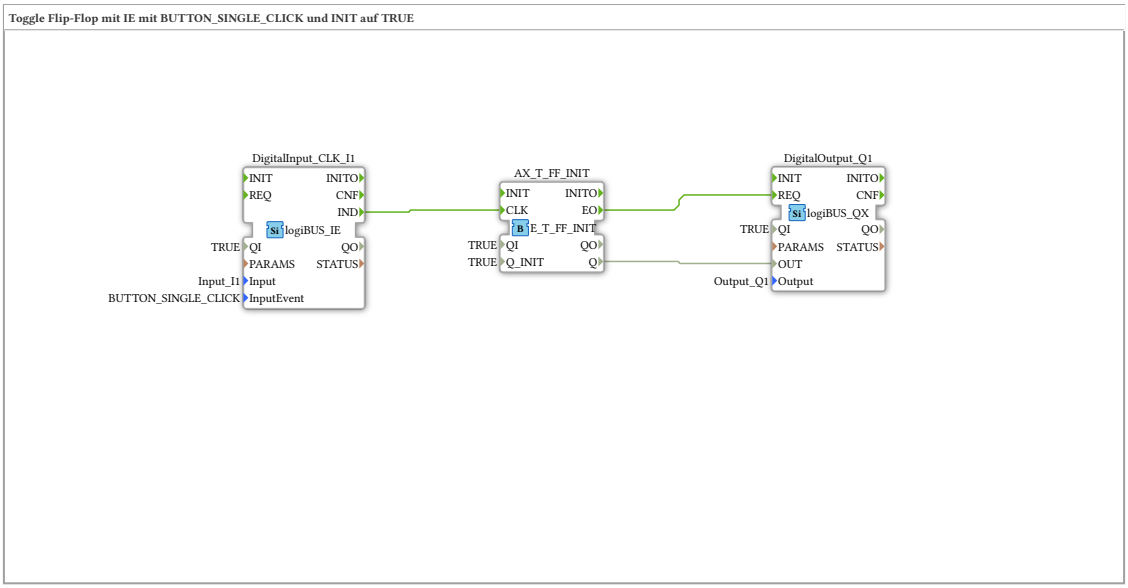
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_004a10a demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.2.1.37 Uebung_004a10b: Toggle Flip-Flop mit IE mit BUTTON_SINGLE_CLICK und INIT auf TRUE

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_004a10b (Toggle Flip-Flop mit IE mit BUTTON_SINGLE_CLICK und INIT auf TRUE).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Toggle Flip-Flop mit IE mit BUTTON_SINGLE_CLICK und INIT auf TRUE**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_004a10b.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QX.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
- **DigitalInput_CLK_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **AX_T_FF_INIT**: Instanz des Typs iec61499::events::E_T_FF_INIT.
  - Parameter QI = TRUE
  - Parameter Q_INIT = TRUE

Verbindungen und Schnittstellen

Ereignisverbindungen:

- DigitalInput_CLK_I1.IND -> AX_T_FF_INIT.CLK
- AX_T_FF_INIT.EO -> DigitalOutput_Q1.REQ

Datenverbindungen:

- AX_T_FF_INIT.Q -> DigitalOutput_Q1.OUT

Hinweise aus dem Modell

am Anfang gleich mal AN !

Programmablauf und Funktionsweise

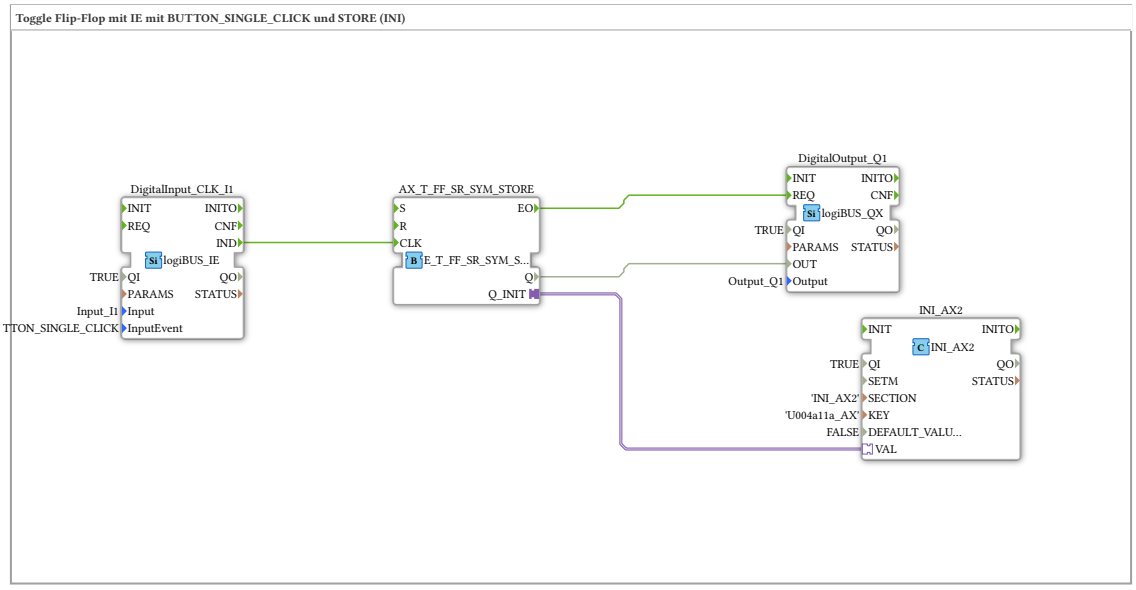
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_004a10b demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.2.1.38 Uebung_004a11a: Toggle Flip-Flop mit IE mit BUTTON_SINGLE_CLICK und STORE (INI)

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_004a11a (Toggle Flip-Flop mit IE mit BUTTON_SINGLE_CLICK und STORE (INI)).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Toggle Flip-Flop mit IE mit BUTTON_SINGLE_CLICK und STORE (INI)**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_004a11a.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QX.

  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
- **DigitalInput_CLK_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **AX_T_FF_SR_SYM_STORE**: Instanz des Typs adapter::iec61499::events::E_T_FF_SR_SYM_STORE.
- **INI_AX2**: Instanz des Typs eclipse4diac::storage::INI_AX2.
  - Parameter QI = TRUE
  - Parameter SECTION = ‚INI_AX2‘
  - Parameter KEY = ‚U004a11a_AX‘
  - Parameter DEFAULT_VALUE = FALSE

Verbindungen und Schnittstellen

Adapterverbindungen:

- AX_T_FF_SR_SYM_STORE.Q_INIT -> INI_AX2.VAL

Ereignisverbindungen:

- DigitalInput_CLK_I1.IND -> AX_T_FF_SR_SYM_STORE.CLK
- AX_T_FF_SR_SYM_STORE.EO -> DigitalOutput_Q1.REQ

Datenverbindungen:

- AX_T_FF_SR_SYM_STORE.Q -> DigitalOutput_Q1.OUT

Hinweise aus dem Modell

am Anfang letzten Zustand laden!

Programmablauf und Funktionsweise

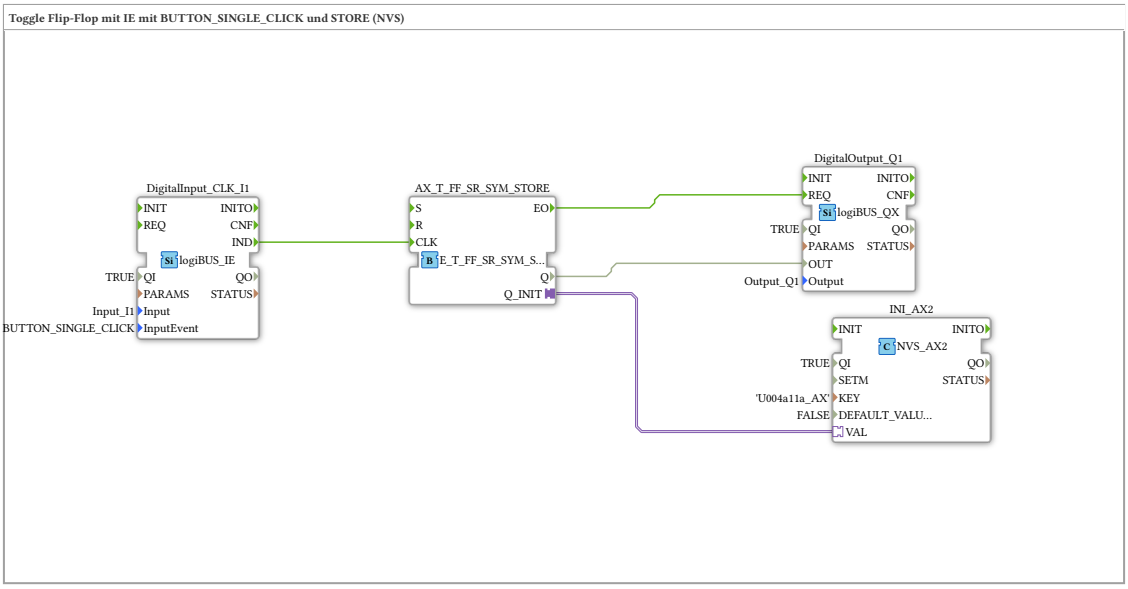
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_004a11a demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.2.1.39 Uebung_004a11b: Toggle Flip-Flop mit IE mit BUTTON_SINGLE_CLICK und STORE (NVS)

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_004a11b (Toggle Flip-Flop mit IE mit BUTTON_SINGLE_CLICK und STORE (NVS)).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Toggle Flip-Flop mit IE mit BUTTON_SINGLE_CLICK und STORE (NVS)**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_004a11b.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QX.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
- **DigitalInput_CLK_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **AX_T_FF_SR_SYM_STORE**: Instanz des Typs adapter::iec61499::events::E_T_FF_SR_SYM_STORE.
- **INI_AX2**: Instanz des Typs logiBUS::storage::esp32_nvs::NVS_AX2.
  - Parameter QI = TRUE
  - Parameter KEY = ‚U004a11a_AX‘
  - Parameter DEFAULT_VALUE = FALSE

Verbindungen und Schnittstellen

Adapterverbindungen:

- AX_T_FF_SR_SYM_STORE.Q_INIT -> INI_AX2.VAL

Ereignisverbindungen:

- DigitalInput_CLK_I1.IND -> AX_T_FF_SR_SYM_STORE.CLK
- AX_T_FF_SR_SYM_STORE.EO -> DigitalOutput_Q1.REQ

Datenverbindungen:

- AX_T_FF_SR_SYM_STORE.Q -> DigitalOutput_Q1.OUT

Hinweise aus dem Modell

am Anfang letzten Zustand laden!

Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_004a11b demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.2.1.40 Uebung_004a2: Toggle Flip-Flop mit IE mit BUTTON_SINGLE_CLICK mit E_MERGE



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_004a2`. Hier wird die Stromstoßschaltung erweitert, sodass sie von zwei verschiedenen Tastern aus bedient werden kann. Dazu werden die Ereignisse der beiden Taster logisch zusammengeführt.

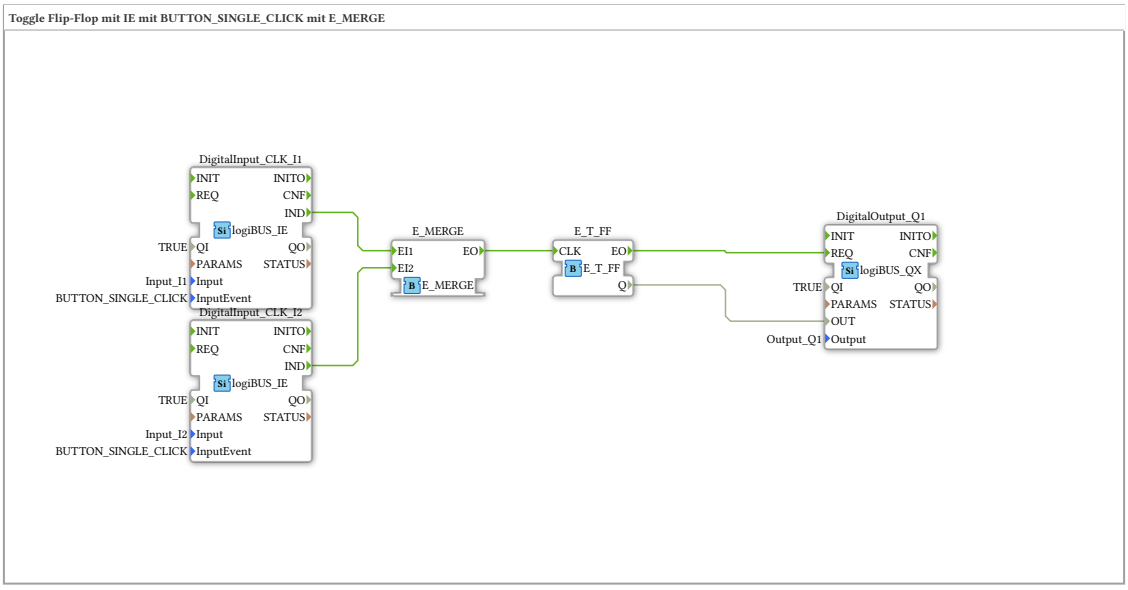
Ziel der Übung

Das Ziel ist es zu lernen, wie man asynchrone Ereignisströme vereint. Wenn zwei verschiedene Ereignisquellen (Taster) denselben Prozess (Licht umschalten) auslösen sollen, müssen ihre Signale „gemerged“ (zusammengeführt) werden, bevor sie den Flip-Flop-Baustein erreichen.

Beschreibung und Komponenten

Die Subapplikation `Uebung_004a2.SUB` nutzt einen `E_MERGE` Baustein, um zwei Eingangs-Events auf einen gemeinsamen Takteingang zu leiten.

Funktionsbausteine (FBs)



- **`DigitalInput_CLK_I1 & I2`**: Zwei `logiBUS_IE` Bausteine, konfiguriert auf `BUTTON_SINGLE_CLICK`. Sie erzeugen Ereignisse bei Betätigung von Taster 1 oder 2.
- **`E_MERGE`**: Ein Standard-Ereignis-Baustein. Er besitzt zwei Ereigniseingänge (`EI1`, `EI2`) und einen Ereignisausgang (`EO`). Jedes eintreffende Event wird sofort an den Ausgang weitergereicht.
- **`E_T_FF`**: Das Toggle-Flip-Flop zum Speichern des Zustands.
- **`DigitalOutput_Q1`**: Der Hardware-Ausgang für die Lampe.

Funktionsweise

Die Verschaltung sorgt für ein logisches ODER der Auslöser:

```
<EventConnections>
    <Connection Source="DigitalInput_CLK_I1.IND" Destination="E_MERGE.EI1"/>
    <Connection Source="DigitalInput_CLK_I2.IND" Destination="E_MERGE.EI2"/>
    <Connection Source="E_MERGE.EO" Destination="E_T_FF.CLK"/>
</EventConnections>
```

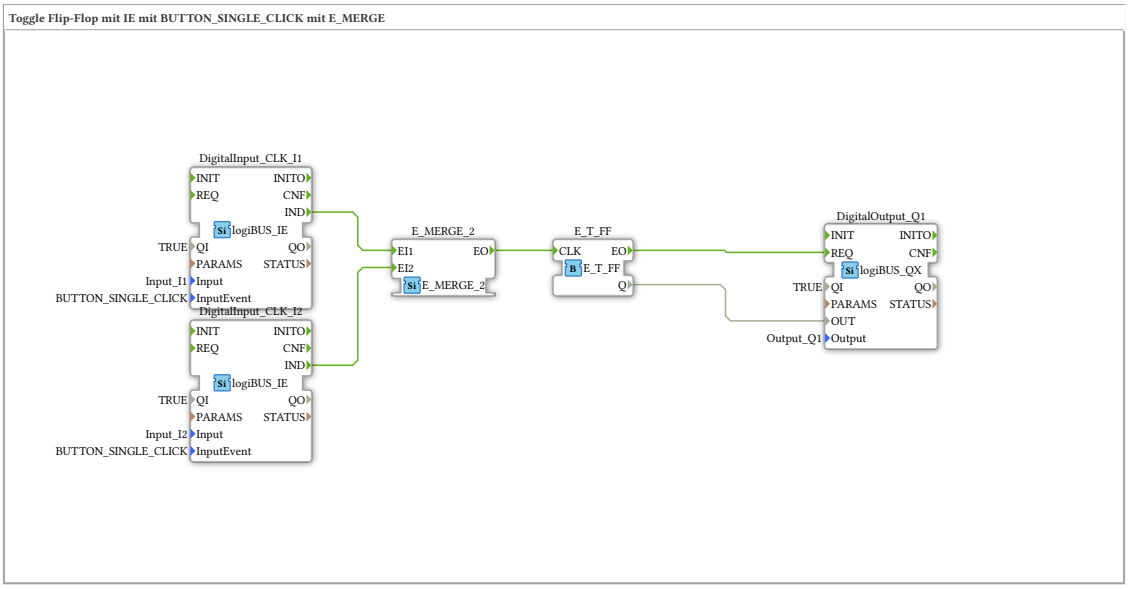
1. Drückt man Taster 1, sendet `I1` ein Event an `E_MERGE.EI1`. `E_MERGE` leitet es an `EO` weiter -> `E_T_FF` toggelt.
2. Drückt man Taster 2, sendet `I2` ein Event an `E_MERGE.EI2`. `E_MERGE` leitet es an `EO` weiter -> `E_T_FF` toggelt.

Das Licht wechselt also bei jedem Klick, egal an welcher Stelle gedrückt wurde.

Anwendungsbeispiel

Wechselschaltung im Flur: Man kann das Licht an einem Ende des Flurs einschalten und am anderen Ende wieder ausschalten. Jeder Tastendruck bewirkt lediglich eine Zustandsänderung („Toggle“), unabhängig davon, wie der aktuelle Zustand ist.

2.2.1.41 Uebung_004a2_2: Toggle Flip-Flop mit IE mit BUTTON_SINGLE_CLICK mit E_MERGE



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung implementiert eine Schaltung, bei der ein Ausgang (Q1) durch zwei verschiedene Eingänge (I1 und I2) umgeschaltet (getoggelt) werden kann. Dabei wird spezifisch auf das „Single Click“ Ereignis der Taster reagiert. Die Logik nutzt ein Event-Toggle-Flip-Flop (`E_T_FF`) in Kombination mit einem Event-Merge-Baustein (`E_MERGE_2`), um die Signale der beiden Taster zusammenzuführen.

Dies entspricht einer klassischen Wechselschaltung oder Stromstoßschaltung in der Gebäudeautomation, realisiert durch Event-Logik in IEC 61499.

Verwendete Funktionsbausteine (FBs)

In dieser SubApp werden verschiedene Funktionsbausteine verschaltet, um die gewünschte Logik zu realisieren.

Sub-Bausteine

DigitalInput_CLK_I1 & DigitalInput_CLK_I2

- **Typ**: `logiBUS::io::DI::logiBUS_IE`

- **Beschreibung**: Diese Bausteine dienen als Schnittstelle zu den physischen Eingängen. Sie sind so konfiguriert, dass sie Ereignisse (Events) generieren.
- **Konfiguration**:
  - **Parameter**: `Input` = `Input_I1` (bzw. `Input_I2`)
  - **Parameter**: `InputEvent` = `BUTTON_SINGLE_CLICK`
- **Funktionsweise**: Der Baustein überwacht den physischen Eingang. Wenn ein einfacher Tastendruck (Single Click) erkannt wird, wird das `IND` Ereignis ausgelöst.

E_MERGE_2

- **Typ**: `iec61499::events::E_MERGE_2`
- **Beschreibung**: Ein Baustein zum Zusammenführen von Ereignisflüssen.
- **Ereigniseingänge**: `EI1`, `EI2`
- **Ereignisausgang**: `EO`
- **Funktionsweise**: Dieser Baustein fungiert als ODER-Glied für Events. Egal, ob ein Ereignis am Eingang `EI1` oder am Eingang `EI2` eintrifft, es wird sofort an den Ausgang `EO` weitergeleitet.

E_T_FF

- **Typ**: `iec61499::events::E_T_FF`
- **Beschreibung**: Ein eventgesteuertes Toggle-Flip-Flop.
- **Ereigniseingang**: `CLK`
- **Datenausgang**: `Q`
- **Ereignisausgang**: `EO`
- **Funktionsweise**: Bei jedem Eintreffen eines Events am `CLK`-Eingang wechselt der boolesche Zustand des Ausgangs `Q` (von FALSE auf TRUE oder umgekehrt). Nach dem Zustandswechsel wird ein Event am `EO` Ausgang gefeuert.

DigitalOutput_Q1

- **Typ**: `logiBUS::io::DQ::logiBUS_QX`
- **Beschreibung**: Schnittstelle zum physischen Ausgang.
- **Konfiguration**:
  - **Parameter**: `Output` = `Output_Q1`
- **Funktionsweise**: Dieser Baustein schreibt den Wert am Eingang `OUT` auf den physischen Ausgang, sobald am Eingang `REQ` ein Event eintrifft.

Programmablauf und Verbindungen

Der Ablauf der Schaltung ist wie folgt definiert:

1. **Eingabeerfassung**:
   - Der Benutzer betätigt entweder den Taster an Eingang `Input_I1` oder an `Input_I2`.
   - Die entsprechenden Bausteine (`DigitalInput_CLK_I1` oder `DigitalInput_CLK_I2`) erkennen den „Single Click“ und senden ein `IND` Event.
2. **Zusammenführung (Merge)**:
   - Das Event von `I1` (verbunden mit `E_MERGE_2.EI1`) oder das Event von `I2` (verbunden mit `E_MERGE_2.EI2`) erreicht den Merge-Baustein.
   - Der `E_MERGE_2` leitet das Event über `EO` weiter an das Flip-Flop.
3. **Verarbeitung (Toggle)**:
   - Das Event erreicht den Takteingang `CLK` des `E_T_FF`.
   - Das Flip-Flop negiert seinen aktuellen Zustand am Datenausgang `Q`.
   - Das Flip-Flop signalisiert den neuen Wert über das Event `EO`.
4. **Ausgabe**:
   - Das Event `EO` des Flip-Flops aktiviert den Ausgangsbaustein `DigitalOutput_Q1` am Eingang `REQ`.
   - Gleichzeitig wird der neue Zustand `Q` an `DigitalOutput_Q1.OUT` übertragen.
   - Die physische Lampe/Aktor an `Output_Q1` schaltet sich ein oder aus.

Zusammenfassung

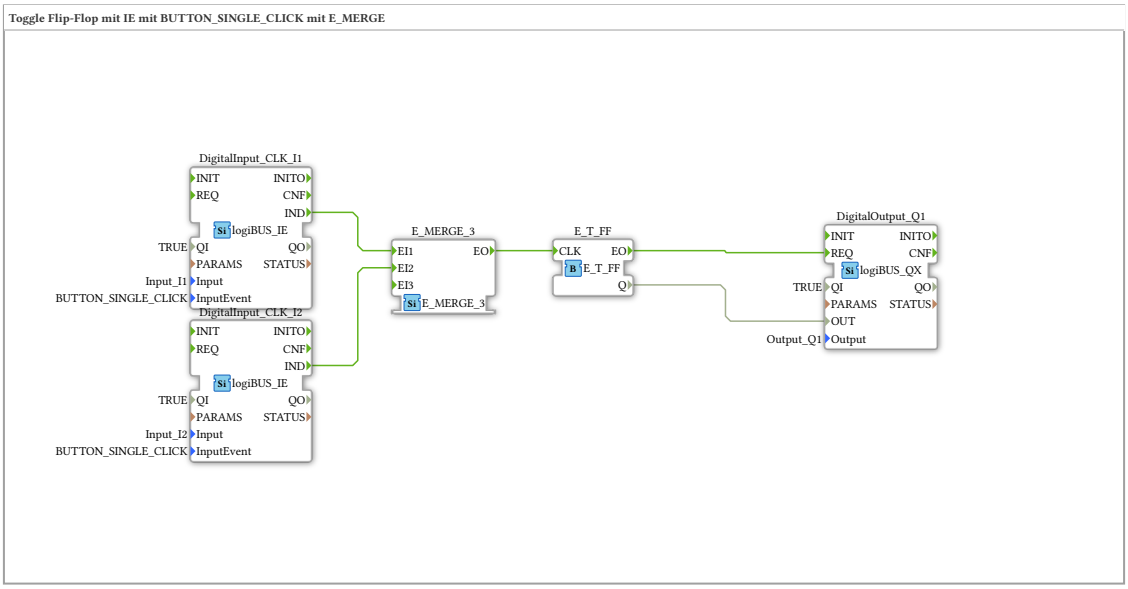
Die Übung `Uebung_004a2_2` demonstriert effektiv, wie man in 4diac mehrere Eingangssignale auf eine gemeinsame Verarbeitungslogik führt. Durch die Verwendung des `E_MERGE_2` Bausteins können sowohl

Taster 1 als auch Taster 2 denselben `E_T_FF` (Toggle Flip-Flop) ansteuern. Dies ermöglicht eine flexible Steuerung eines Ausgangs von mehreren Orten aus, ähnlich einer Stromstoßschaltung in der Elektrotechnik.

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

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

2.2.1.42 Uebung_004a2_3: Toggle Flip-Flop mit IE mit BUTTON_SINGLE_CLICK mit E_MERGE



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung implementiert eine **Stromstoßschaltung** (Toggle-Funktion), die von zwei verschiedenen Eingängen aus bedient werden kann. Das Ziel ist es, einen digitalen Ausgang (z.B. eine Lampe) durch Betätigen eines von zwei Tastern an- und wieder auszuschalten.

Hierbei kommt ein Toggle-Flip-Flop (`E_T_FF`) zum Einsatz, welches seinen Zustand bei jedem Eingangsimpuls wechselt. Die Besonderheit liegt in der Verwendung von Event-Eingangsbausteinen (`logiBUS_IE`), die spezifisch auf einen „Single Click“ (`BUTTON_SINGLE_CLICK`) reagieren, sowie einem Merge-Baustein (`E_MERGE`), der die Signale der beiden Taster zusammenführt.

Verwendete Funktionsbausteine (FBs)

In dieser Sub-Applikation werden verschiedene Bausteine aus der `logiBUS` und der `iec61499` Standardbibliothek verwendet, um die Logik zu realisieren.

Sub-Bausteine: DigitalInput_CLK_I1 & DigitalInput_CLK_I2

Die Eingangsbausteine erfassen die Tasterbetätigung.

- **Typ**: `logiBUS::io::DI::logiBUS_IE`
- **Verwendete interne FBs**:
  - **DigitalInput_CLK_I1**:
    - Parameter: `Input` = `Input_I1`
    - Parameter: `InputEvent` = `BUTTON_SINGLE_CLICK`
    - Ereignisausgang: `IND` (Indication - Signalisiert den Klick)
  - **DigitalInput_CLK_I2**:
    - Parameter: `Input` = `Input_I2`
    - Parameter: `InputEvent` = `BUTTON_SINGLE_CLICK`
    - Ereignisausgang: `IND`

- **Funktionsweise**: Diese Bausteine überwachen die physischen Eingänge I1 und I2. Sie sind so konfiguriert, dass sie nur bei einem einfachen Klick (`BUTTON_SINGLE_CLICK`) ein Event am Ausgang `IND` erzeugen.

Sub-Bausteine: E_MERGE_3

Dient der Zusammenführung von Event-Strömen.

- **Typ**: `iec61499::events::E_MERGE_3`
- **Verwendete interne FBs**:
  - **E_MERGE_3**:
    - Ereigniseingang: `EI1` (Verbunden mit I1)
    - Ereigniseingang: `EI2` (Verbunden mit I2)
    - Ereignisausgang: `EO`
- **Funktionsweise**: Der Merge-Baustein fungiert als ODER-Glied für Events. Egal ob das Signal von Taster I1 oder Taster I2 kommt, das Event wird an den Ausgang `EO` durchgereicht.

Sub-Bausteine: E_T_FF

Das eigentliche Speicherelement der Schaltung.

- **Typ**: `iec61499::events::E_T_FF`
- **Verwendete interne FBs**:
  - **E_T_FF**:
    - Ereigniseingang: `CLK` (Clock)
    - Datenausgang: `Q` (Aktueller Status: TRUE/FALSE)
    - Ereignisausgang: `EO` (Event Output nach Zustandswechsel)
- **Funktionsweise**: Das Toggle-Flip-Flop wechselt bei jedem eingehenden Event am `CLK`-Eingang seinen internen Zustand `Q` (von 0 auf 1 oder von 1 auf 0).

Sub-Bausteine: DigitalOutput_Q1

Stellt die Verbindung zur physischen Hardware her.

- **Typ**: `logiBUS::io::DQ::logiBUS_QX`
- **Verwendete interne FBs**:
  - **DigitalOutput_Q1**:
    - Parameter: `Output` = `Output_Q1`
    - Dateneingang: `OUT` (Verbunden mit T_FF.Q)
    - Ereigniseingang: `REQ` (Request)
- **Funktionsweise**: Dieser Baustein schreibt den logischen Zustand auf den physischen Ausgang Q1.

Programmablauf und Verbindungen

Der Ablauf der Schaltung gestaltet sich wie folgt:

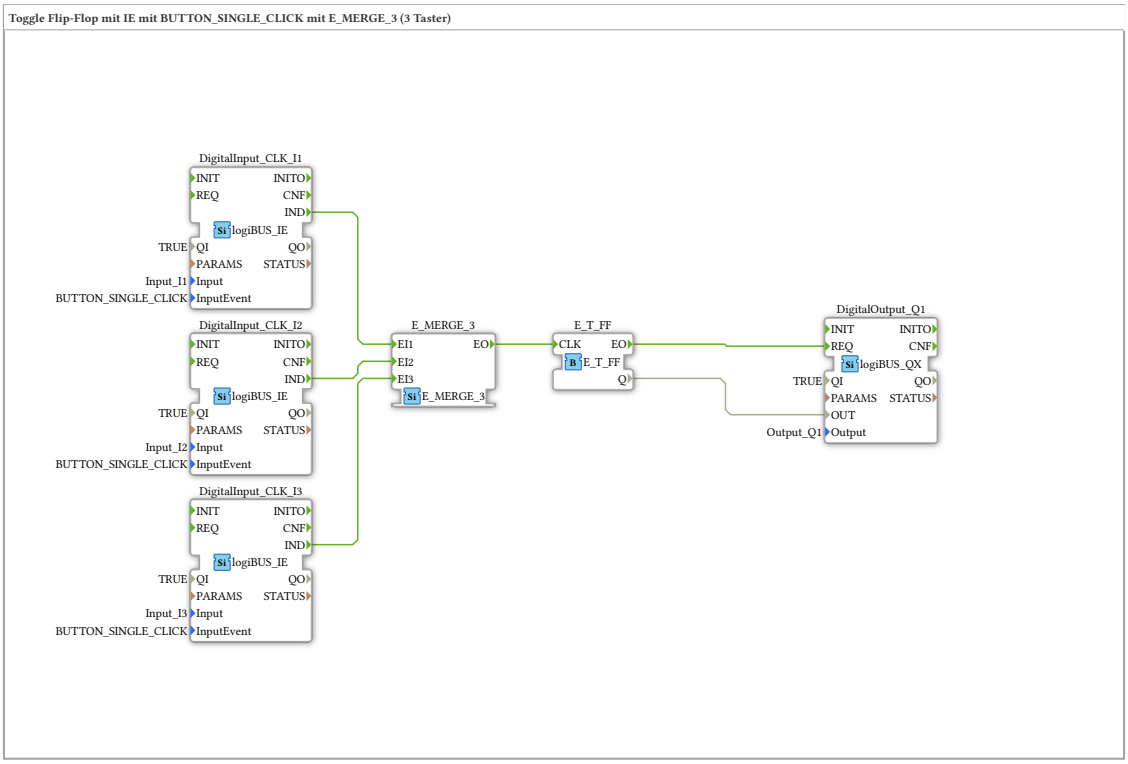
1. **Eingabeerfassung**: Der Benutzer betätigt entweder Taster an Eingang `Input_I1` oder an Eingang `Input_I2`. Die Bausteine `DigitalInput_CLK_I1` bzw. `_I2` erkennen einen einzelnen Klick (`BUTTON_SINGLE_CLICK`).
2. **Signalevent**: Ein Event (`IND`) wird vom betätigten Eingangsbaustein ausgesendet.
3. **Zusammenführung (Merge)**: Die Events beider Eingänge sind mit dem `E_MERGE_3` Baustein verbunden (an `EI1` und `EI2`). Sobald eines der Events eintrifft, gibt der Merge-Baustein sofort ein Event am Ausgang `EO` aus.
4. **Umschaltung (Toggle)**: Das zusammengeführte Event erreicht den `CLK`-Eingang des `E_T_FF`. Dies veranlasst das Flip-Flop, seinen Zustand `Q` zu invertieren (umzuschalten).
5. **Ausgabe**:
   - Das Daten-Signal `Q` (TRUE/FALSE) wird an den Dateneingang `OUT` des Ausgangsbausteins `DigitalOutput_Q1` gesendet.
   - Gleichzeitig triggert der Event-Ausgang `EO` des Flip-Flops den `REQ`-Eingang des Ausgangsbausteins, um die Aktualisierung des physikalischen Ausgangs durchzuführen.

Zusammenfassung

In der Übung `Uebung_004a2_3` wird eine klassische Tasterschaltung mit zwei Bedienstellen realisiert. Lernziele sind der Umgang mit dem `E_MERGE`-Baustein zur Bündelung von Eventsignalen und die Anwen-

dung des `E_T_FF` (Toggle-Flip-Flop) zur Zustandsspeicherung. Zudem wird verdeutlicht, wie spezifische Taster-Events (hier: Single Click) in der LogiBUS-Bibliothek verarbeitet werden.

2.2.1.43 Uebung_004a2_3b: Toggle Flip-Flop mit IE mit BUTTON_SINGLE_CLICK mit E_MERGE_3 (3 Taster)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung realisiert ein **Toggle Flip-Flop** (auch als **T-Flipflop** bekannt).
Durch einen Tastendruck (Einzelfunktion – BUTTON_SINGLE_CLICK) an einem von drei Tastern (I1, I2 oder I3) wird der angeschlossene Ausgang Q1 jeweils umgeschaltet (ein-/ausgeschaltet).
Die drei Taster werden über einen **E_MERGE_3**-Baustein zusammengeführt, sodass jeder beliebige Tastendruck das Flipflop triggert.

Verwendete Funktionsbausteine (FBs)

Die Übung besteht ausschließlich aus primitiven (vordefinierten) Funktionsbausteinen. Es werden keine eigenen Unterbausteine (SubApps) verwendet.

Primitive Funktionsbausteine

- **DigitalInput_CLK_I1** (Typ: `logiBUS::io::DI::logiBUS_IE`)
  - Parametereinstellungen:
    - `QI` = `TRUE`
    - `Input` = `Input_I1`
    - `InputEvent` = `BUTTON_SINGLE_CLICK`
  - Ereignisausgang: `IND` (wird bei Tastendruck ausgelöst)
  - Datenausgang: wird nicht verwendet (keine Datenverbindung)
- **DigitalInput_CLK_I2** (Typ: `logiBUS::io::DI::logiBUS_IE`)
  - Parametereinstellungen:
    - `QI` = `TRUE`
    - `Input` = `Input_I2`
    - `InputEvent` = `BUTTON_SINGLE_CLICK`

  - Ereignisausgang: `IND`
- **DigitalInput_CLK_I3** (Typ: `logiBUS::io::DI::logiBUS_IE`)
  - Parametereinstellungen:
    - `QI` = `TRUE`
    - `Input` = `Input_I3`
    - `InputEvent` = `BUTTON_SINGLE_CLICK`
  - Ereignisausgang: `IND`
- **E_MERGE_3** (Typ: `iec61499::events::E_MERGE_3`)
  - Ereigniseingänge: `EI1`, `EI2`, `EI3`
  - Ereignisausgang: `EO` (wird ausgelöst, sobald eines der Eingangsereignisse eintrifft)
- **E_T_FF** (Typ: `iec61499::events::E_T_FF`)
  - Ereigniseingang: `CLK` (Takt – schaltet bei jedem eingehenden Ereignis um)
  - Ereignisausgang: `EO` (wird bei jedem Toggeln ausgelöst)
  - Datenausgang: `Q` (logischer Zustand, TRUE oder FALSE)
- **DigitalOutput_Q1** (Typ: `logiBUS::io::DQ::logiBUS_QX`)
  - Parametereinstellungen:
    - `QI` = `TRUE`
    - `Output` = `Output_Q1`
  - Ereigniseingang: `REQ` (Anforderung zum Setzen des Ausgangs)
  - Dateneingang: `OUT` (Wert, der auf den reellen Ausgang geschrieben wird)

Programmablauf und Verbindungen

1. **Ereignisverkettung**
   - Die drei Tasterbausteine (`DigitalInput_CLK_I1`, `I2`, `I3`) erzeugen bei einem Tastendruck (BUTTON_SINGLE_CLICK) ein Ereignis an ihrem Ausgang `IND`.
   - Diese drei Ereignisse sind mit den drei Eingängen des `E_MERGE_3` (EI1 ... EI3) verbunden.
   - Sobald einer der drei Taster gedrückt wird, erzeugt `E_MERGE_3` ein Ereignis an seinem Ausgang `EO`.
   - Dieses Ereignis wird an den Takteingang `CLK` des `E_T_FF` weitergeleitet.
   - Das `E_T_FF` toggelt bei jedem Ereignis: Sein Ausgang `Q` wechselt zwischen TRUE und FALSE. Gleichzeitig wird das Ereignis `EO` des Flipflops ausgelöst.
2. **Datenverkettung**
   - Der Zustand des Flipflops (Datenausgang `Q`) wird direkt an den Dateneingang `OUT` des Ausgangsbausteins `DigitalOutput_Q1` geführt.
   - Das Ereignis `EO` des Flipflops löst den `REQ`-Eingang des Ausgangsbausteins aus, sodass der aktuelle Wert auf den physikalischen Ausgang **Output_Q1** geschrieben wird.
3. **Zusammenfassung des Ablaufs**
   - Jeder einzelne Tastendruck (an I1, I2 oder I3) führt zu einer Umschaltung des Ausgangs.
   - Dadurch kann der Ausgang mit drei verschiedenen Tastern gesteuert werden (Toggle-Funktion).

Zusammenfassung

Diese Übung veranschaulicht die **Kombination von Ereignis- und Datenflüssen** in IEC 61499:

- Drei gleichberechtigte Taster werden über einen **E_MERGE_3** zu einem gemeinsamen Ereignis zusammengeführt.
- Ein **E_T_FF** (Toggle-Flipflop) realisiert die eigentliche Umschaltlogik.
- Der Ausgangsbaustein setzt den physikalischen Ausgang entsprechend dem Flipflop-Zustand.

Lernziele:

- Verständnis des Zusammenspiels mehrerer ereignisgesteuerter Eingänge.
- Anwendung eines T-Flipflops als einfachen Zustandsspeicher.
- Praxisnahe Nutzung von logiBUS-Ein- und Ausgangsbausteinen mit BUTTON_SINGLE_CLICK-Ereignissen.

Schwierigkeitsgrad: Einsteiger
Voraussetzungen: Grundlegende Kenntnisse der IEC 61499-Ereignissteuerung.

2.2.1.44 Uebung_004a3: wie Uebung_004a2 aber ohne E_MERGE

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_004a3`. Diese Übung zeigt eine Vereinfachung gegenüber `Uebung_004a2`: In IEC 61499 können mehrere Ereignisquellen oft direkt auf denselben Ereigniseingang verbunden werden.

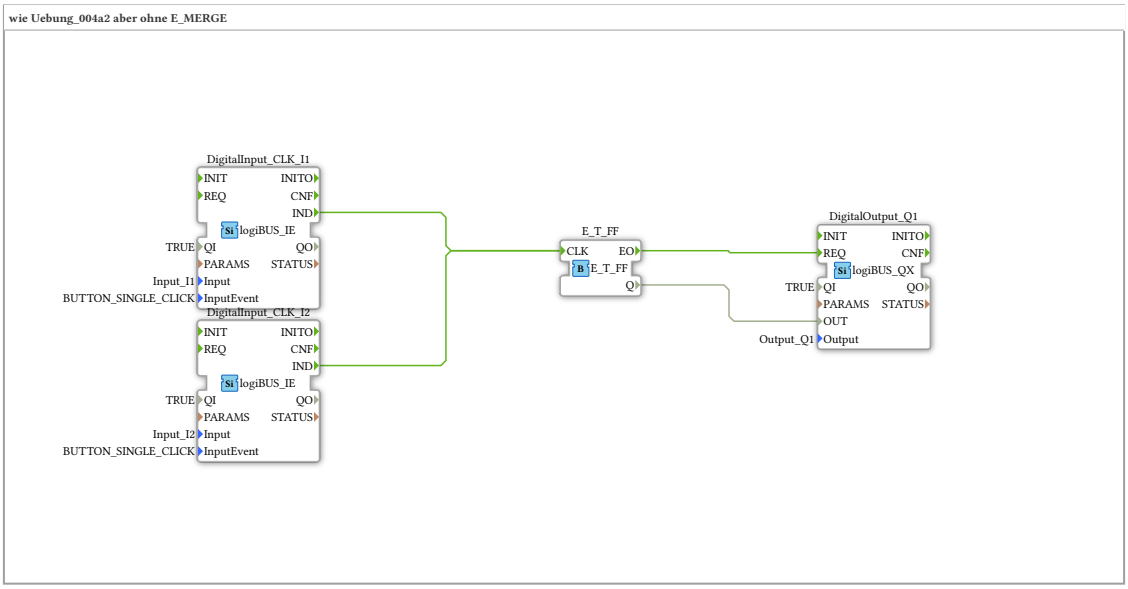
Ziel der Übung

Das Ziel ist die Reduktion der visuellen Komplexität im Netzwerk-Diagramm. Es wird demonstriert, dass der explizite `E_MERGE` Baustein weggelassen werden kann, da die 4diac-Laufzeitumgebung eingehende Events an einem Port automatisch nacheinander verarbeitet („Fan-In“).

Beschreibung und Komponenten

Die Subapplikation `Uebung_004a3.SUB` verbindet zwei Event-Quellen direkt mit dem Takteingang des Flip-Flops.

Funktionsbausteine (FBs)



- **`DigitalInput_CLK_I1` & `I2`**: Die ereignisbasierten Eingänge.
- **`E_T_FF`**: Das Toggle-Flip-Flop.
- **`DigitalOutput_Q1`**: Der Ausgang.

Der Baustein `E_MERGE` aus der vorherigen Übung fehlt hier bewusst.

Funktionsweise

```
<EventConnections>
    <Connection Source="DigitalInput_CLK_I1.IND" Destination="E_T_FF.CLK"/>
    <Connection Source="DigitalInput_CLK_I2.IND" Destination="E_T_FF.CLK"/>
</EventConnections>
```

Die Funktionsweise ist identisch zur Übung mit `E_MERGE`: Jedes eintreffende Event an `E_T_FF.CLK` – egal ob von `I1` oder `I2` kommend – triggert die Ausführung des Funktionsbausteins. 4diac unterstützt diese Mehrfachverbindung für Events nativ.

> **Wichtiger Hinweis:** Bei **Datenverbindungen** ist dies **nicht erlaubt**! Zwei Datenausgänge dürfen niemals direkt auf denselben Dateneingang schreiben, da dies zu Konflikten führen würde. Bei Events hingegen ist dies eine effiziente Methode für „ODER“-Verknüpfungen von Auslösern.

Anwendungsbeispiel

Gleiches Beispiel wie zuvor (Wechselschaltung), jedoch mit schlankerem Code (weniger Bausteine, höhere Übersichtlichkeit).

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

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

2.2.1.45 Uebung_004a4: Toggle Flip-Flop mit IE mit BUTTON_SINGLE_CLICK mit E_SPLIT



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_004a4`. Hier wird gezeigt, wie ein einzelnes Ereignis genutzt werden kann, um mehrere unabhängige Prozesse sequenziell anzustoßen, indem man einen `E_SPLIT` Baustein verwendet.

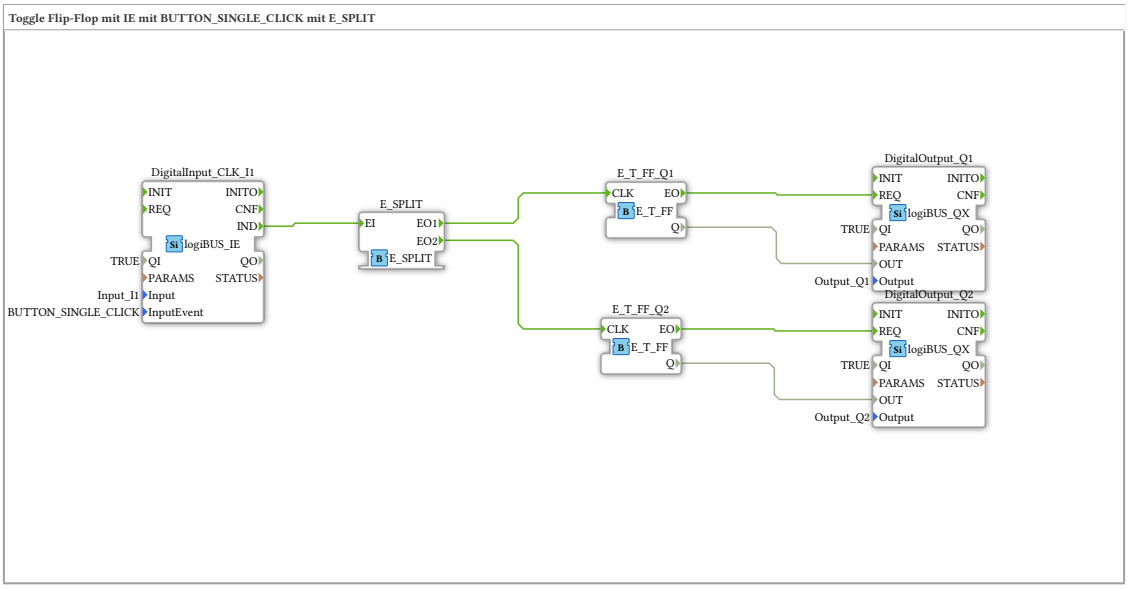
Ziel der Übung

Das Ziel ist das Verständnis der sequenziellen Ereignis-Verarbeitung. Der `E_SPLIT` Baustein nimmt ein einzelnes Eingangs-Event entgegen und feuert daraufhin seine Ausgänge nacheinander ab. Dies ermöglicht es, eine Aktion an mehrere Ziele zu verteilen und dabei die Reihenfolge der Abarbeitung festzulegen.

Beschreibung und Komponenten

Die Subapplikation `Uebung_004a4.SUB` verwendet einen Taster, um zwei separate Toggle-Flip-Flops gleichzeitig zu schalten.

Funktionsbausteine (FBs)



- **`DigitalInput_CLK_I1`**: Der Event-Generator (Klick-Taster).
- **`E_SPLIT`**: Ein Ereignis-Verteiler. Er hat einen Eingang `EI` und zwei Ausgänge `EO1` und `EO2`.
- **`E_T_FF_Q1`** & **`E_T_FF_Q2`**: Zwei unabhängige Flip-Flops.
- **`DigitalOutput_Q1`** & **`DigitalOutput_Q2`**: Zwei physische Ausgänge.

Funktionsweise

```
<EventConnections>
    <Connection Source="DigitalInput_CLK_I1.IND" Destination="E_SPLIT.EI"/>
    <Connection Source="E_SPLIT.EO1" Destination="E_T_FF_Q1.CLK"/>
```

```
    <Connection Source="E_SPLIT.EO2" Destination="E_T_FF_Q2.CLK"/>
</EventConnections>
```

1. Ein Klick auf Taster 1 sendet ein Event an `E_SPLIT.EI`.
2. `E_SPLIT` sendet daraufhin **zuerst** ein Event an `EO1` ➡️ `E_T_FF_Q1` schaltet um.
3. Unmittelbar danach sendet `E_SPLIT` ein Event an `EO2` ➡️ `E_T_FF_Q2` schaltet um.

Beide Lampen wechseln synchron ihren Zustand, gesteuert durch einen einzigen Taster.

> **Hinweis:** Wie im Quellcode vermerkt, wäre es funktional effizienter, beide Ausgänge an ein einziges Flip-Flop zu hängen. Diese Übung dient jedoch rein zur Demonstration der Event-Verteilung mit `E_SPLIT`.

Anwendungsbeispiel

Zentral-Aus Schaltung: Ein Taster „Feierabend“ löst über einen Splitter mehrere Aktionen nacheinander aus: Zuerst wird die Arbeitsbeleuchtung ausgeschaltet (`Q1`) und danach die Stromzufuhr für die Maschinen gekappt (`Q2`).

2.2.1.46 Uebung_004a5: wie Uebung_004a4 aber ohne E_SPLIT

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_004a5`. Ähnlich wie bei der Zusammenführung von Events wird hier gezeigt, dass auch die Verteilung eines Events auf mehrere Ziele oft ohne expliziten Baustein möglich ist.

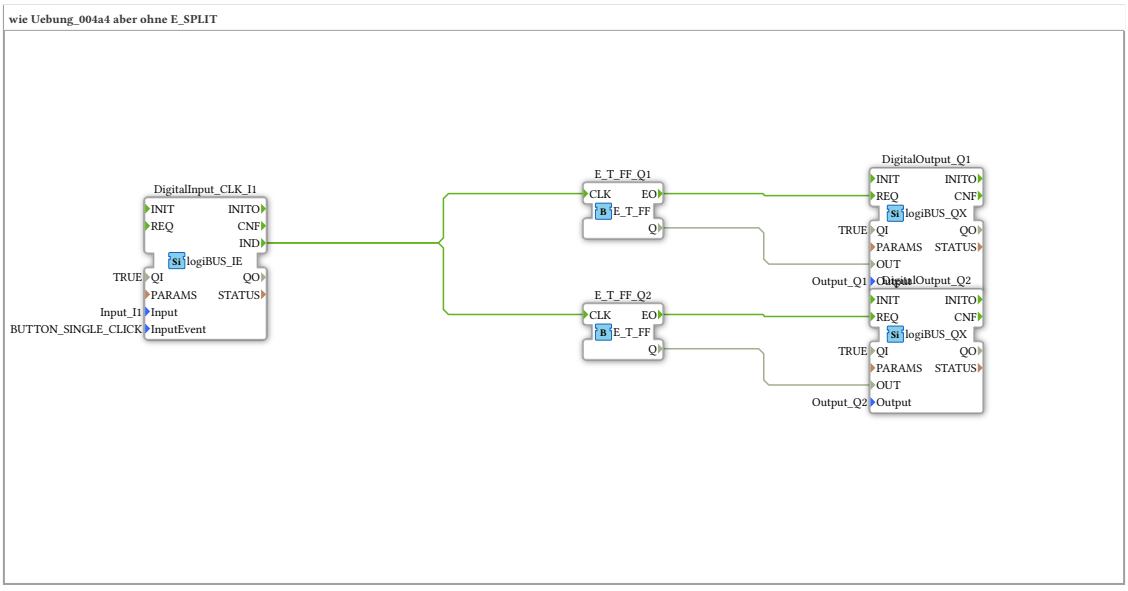
Ziel der Übung

Demonstration der „Fan-Out“-Fähigkeit von Ereignisverbindungen in 4diac. Ein einzelner Ereignis-Ausgang kann mit mehreren Ereignis-Eingängen verbunden werden, um parallele Aktionen auszulösen.

Beschreibung und Komponenten

Die Subapplikation `Uebung_004a5.SUB` entfernt den `E_SPLIT` Baustein aus der vorherigen Übung und verbindet den Taster direkt mit beiden Flip-Flops.

Funktionsbausteine (FBs)



- **`DigitalInput_CLK_I1`**: Taster.

- **`E_T_FF_Q1` & `Q2`**: Zwei unabhängige Flip-Flops.

Funktionsweise

```
<EventConnections>
    <Connection Source="DigitalInput_CLK_I1.IND" Destination="E_T_FF_Q1.CLK"/>
    <Connection Source="DigitalInput_CLK_I1.IND" Destination="E_T_FF_Q2.CLK"/>
</EventConnections>
```

Wenn `I1` ein Ereignis feuert, wird dieses an alle verbundenen Ziele verteilt. Die Reihenfolge der Abarbeitung ist in der IEC 61499 Norm für diesen Fall nicht strikt definiert (meistens erfolgt sie in der Reihenfolge, in der die Verbindungen erstellt wurden).

Wann nutzt man was?

- Nutzen Sie **direkte Verbindungen (Fan-Out)**, wenn die Reihenfolge der Abarbeitung keine Rolle spielt (wie hier beim gleichzeitigen Toggelt zweier Lampen).
- Nutzen Sie einen **`E_SPLIT` Baustein**, wenn eine exakte Abfolge (zuerst A, dann B) technisch zwingend erforderlich ist.

Anwendungsbeispiel

Gleiches Beispiel wie zuvor (Zentral-Aus), jedoch platzsparender implementiert. Dies ist der Standardweg in 4diac, um Signale zu vervielfältigen.

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

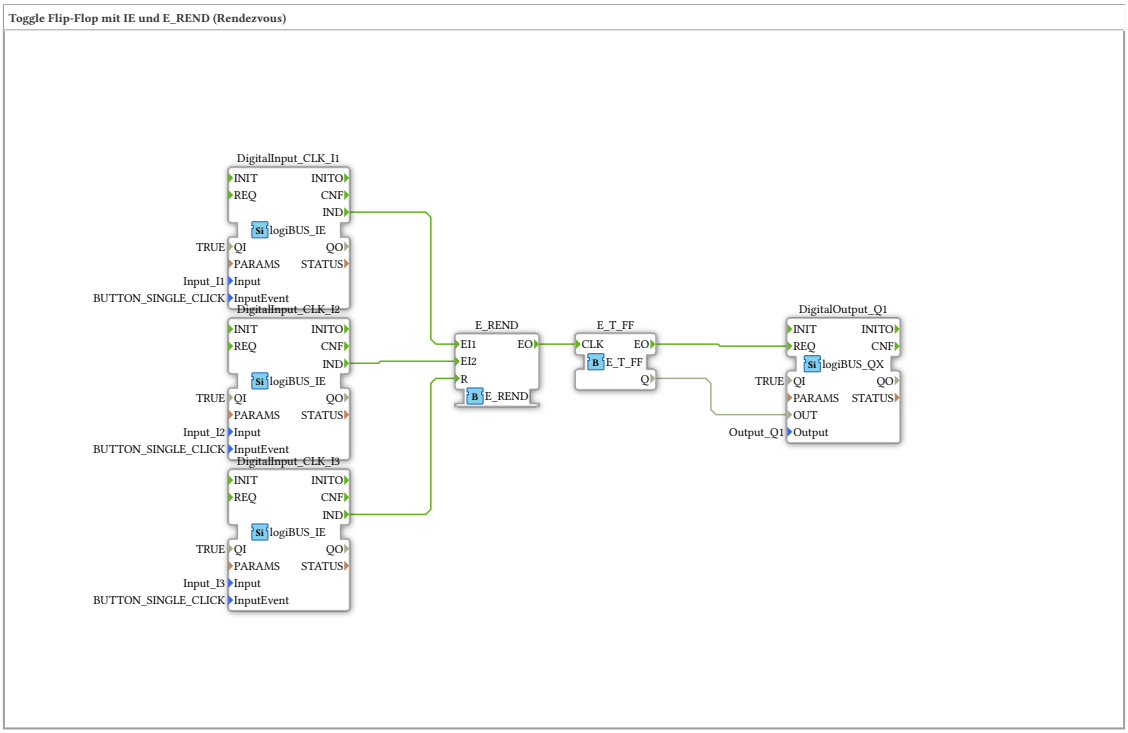
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.2.1.47 Uebung_004a6: Toggle Flip-Flop mit IE und E_REND (Rendezvous)



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_004a6`. Hier wird ein fortgeschrittenes Ereignis-Muster vorgestellt: Das Rendezvous. Ein Ereignis wird erst dann weitergegeben, wenn mehrere unterschiedliche Bedingungen zeitunabhängig eingetroffen sind.

Ziel der Übung



Erlernen des Umgangs mit dem `E_REND` Baustein. Dieser fungiert wie ein „Gedächtnis-UND“ für Ereignisse. Er feuert erst am Ausgang, wenn an *allen* konfigurierten Eingängen mindestens einmal ein Ereignis registriert wurde. Dies dient der Synchronisation von asynchronen Prozessen.

Beschreibung und Komponenten

Die Subapplikation `Uebung_004a6.SUB` nutzt `E_REND`, um sicherzustellen, dass zwei Taster gedrückt wurden, bevor der Ausgang umschaltet.

Funktionsbausteine (FBs)

- **`DigitalInput_CLK_I1` & `I2`**: Die beiden Taster für die Synchronisation.
- **`DigitalInput_CLK_I3`**: Ein Reset-Taster zum Löschen der Vorbedingungen.
- **`E_REND`**: Der Rendezvous-Baustein mit Eingängen `EI1`, `EI2` und einem Reset-Eingang `R`.
- **`E_T_FF`**: Das Flip-Flop zur Zustandsspeicherung.

Funktionsweise

Die Logik verlangt die Bestätigung beider Quellen:

```
<EventConnections>
    <Connection Source="DigitalInput_CLK_I1.IND" Destination="E_REND.EI1"/>
    <Connection Source="DigitalInput_CLK_I2.IND" Destination="E_REND.EI2"/>
    <Connection Source="E_REND.EO" Destination="E_T_FF.CLK"/>
    <Connection Source="DigitalInput_CLK_I3.IND" Destination="E_REND.R"/>
</EventConnections>
```

Der funktionale Ablauf:

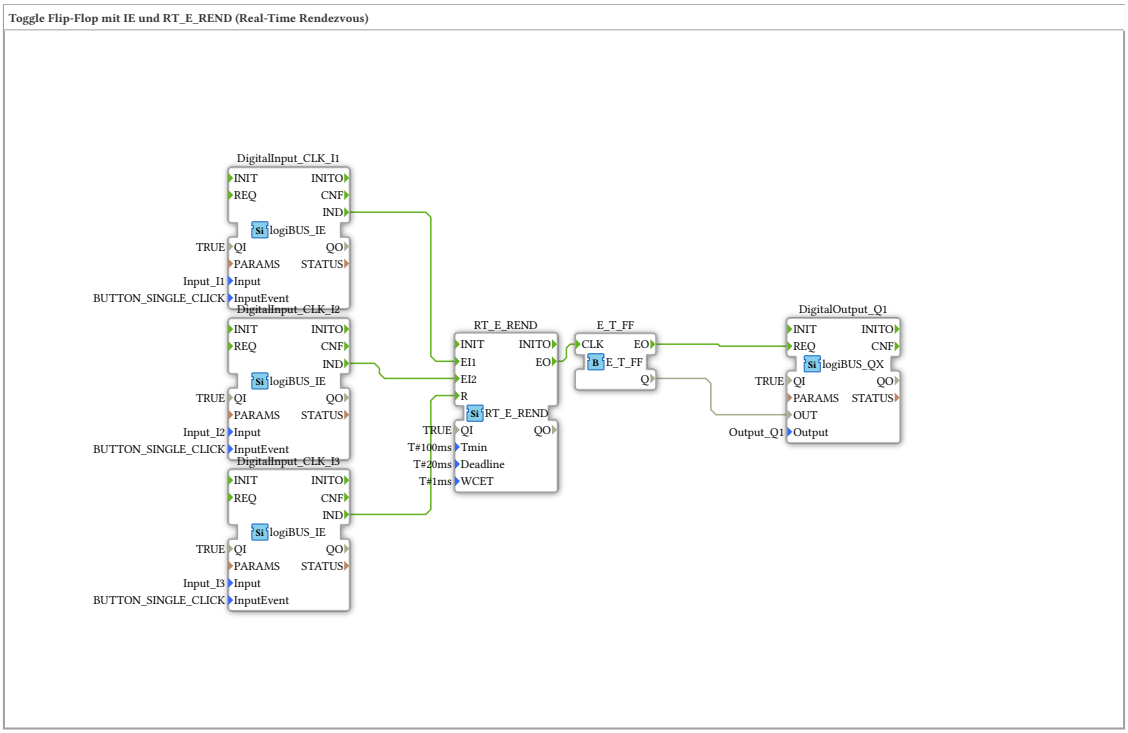
1. Drückt man nur Taster 1 (`I1`), passiert am Ausgang nichts. `E_REND` speichert intern: „EI1 ist erledigt“.
2. Drückt man irgendwann später Taster 2 (`I2`), ist die Bedingung erfüllt (beide waren da). `E_REND` feuert nun das Event an `EO`.
3. Das Flip-Flop toggelt den Ausgangszustand.
4. Danach setzt sich `E_REND` automatisch zurück und wartet erneut auf beide Eingänge.

- Der Reset-Taster (`I3`) kann jederzeit genutzt werden, um die internen Merker von `E_REND` zu löschen (Abbruch der Sequenz).

Anwendungsbeispiel

Sequenzielle Freigabe: In einer Montagehalle muss ein Monteur den Zusammenbau bestätigen (`I1`) und ein Qualitätskontrolleur die Prüfung abnehmen (`I2`). Erst wenn beide (unabhängig voneinander und in beliebiger Reihenfolge) ihre Quittierung gegeben haben, darf das Förderband zum nächsten Schritt weiterschalten (`EO`).

2.2.1.48 Uebung_004a6a: Toggle Flip-Flop mit IE und RT_E_REND (Real-Time Rendezvous)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung implementiert einen Toggle-Flip-Flop, der durch Echtzeit-Ereignisse gesteuert wird. Sie dient dem Kennenlernen des Zusammenwirkens von digitalen Eingängen (IE), einem Real-Time Rendezvous-Baustein (RT_E_REND) und einem Toggle-Flip-Flop (E_T_FF). Die Ausgabe erfolgt auf einen digitalen Ausgang (QX). Ziel ist es, das Verständnis für zeitgesteuerte Ereignisketten in 4diac zu vertiefen.

Verwendete Funktionsbausteine (FBs)

logiBUS_IE (DigitalInput_CLK_I1, DigitalInput_CLK_I2, DigitalInput_CLK_I3)

- **Typ**: logiBUS::io::DI::logiBUS_IE
- **Parameter**:
  - `QI` = `TRUE`
  - `Input` = jeweiliger physischer Eingang (`Input_I1`, `Input_I2`, `Input_I3`)
  - `InputEvent` = `BUTTON_SINGLE_CLICK`
- **Funktionsweise**: Erfasst einen Tastendruck (Single Click) an einem digitalen Eingang und erzeugt ein Ereignis am Ausgangsereignis `IND`. Dient als Startquelle für die Ereigniskette.

RT_E_REND

- **Typ**: eclipse4diac::rtevents::RT_E_REND
- **Parameter**:
  - `QI` = `TRUE`
  - `Tmin` = `T#100ms` (Minimale Zeit zwischen zwei Auslösungen)
  - `Deadline` = `T#20ms` (Maximal zulässige Reaktionszeit)
  - `WCET` = `T#1ms` (Worst-Case Execution Time)

- **Ereigniseingänge**:
  - `EI1` – Erster Starteingang (verbunden mit DigitalInput_CLK_I1.IND)
  - `EI2` – Zweiter Starteingang (verbunden mit DigitalInput_CLK_I2.IND)
  - `R` – Rücksetzeingang (verbunden mit DigitalInput_CLK_I3.IND)
- **Ereignisausgang**:
  - `EO` – Ausgangsereignis (verbunden mit E_T_FF.CLK)
- **Funktionsweise**: Realisiert ein Real-Time Rendezvous. Es wartet auf Ereignisse an `EI1` und `EI2`. Erst wenn beide innerhalb der Deadline (20 ms) eingetroffen sind, wird ein Ereignis am Ausgang `EO` erzeugt. Der Eingang `R` setzt den internen Zustand zurück. Damit wird sichergestellt, dass die nachfolgende Logik nur bei synchronen Ereignissen ausgelöst wird.

E_T_FF

- **Typ**: iec61499::events::E_T_FF
- **Keine Parameter**
- **Ereigniseingang**:
  - `CLK` – Takt-Eingang (verbunden mit RT_E_REND.EO)
- **Datenausgang**:
  - `Q` – Ausgangswert (Bool, verbunden mit DigitalOutput_Q1.OUT)
- **Funktionsweise**: Toggle-Flip-Flop. Bei jedem Ereignis am Takt-Eingang (`CLK`) wird der interne Zustand umgeschaltet. Der Ausgang `Q` gibt den aktuellen Zustand aus (TRUE/FALSE).

logiBUS_QX (DigitalOutput_Q1)

- **Typ**: logiBUS::io::DQ::logiBUS_QX
- **Parameter**:
  - `QI` = `TRUE`
  - `Output` = `Output_Q1` (physischer Ausgang)
- **Ereigniseingang**:
  - `REQ` – Anforderungsereignis (verbunden mit E_T_FF.EO)
- **Dateneingang**:
  - `OUT` – Ausgangswert (verbunden mit E_T_FF.Q)
- **Funktionsweise**: Setzt den physischen digitalen Ausgang `Output_Q1` auf den Wert, der am Dateneingang `OUT` anliegt, sobald ein Ereignis am `REQ`-Eingang eintrifft.

Programmablauf und Verbindungen

Die Übung verwendet drei digitale Eingänge (`I1`, `I2`, `I3`) und einen digitalen Ausgang (`Q1`).

1. **Ereignisverkettung**:
   - Wird an `I1` ein Tastendruck erkannt, sendet `DigitalInput_CLK_I1` ein Ereignis über `IND` an den Ereigniseingang `EI1` von `RT_E_REND`.
   - Wird an `I2` ein Tastendruck erkannt, sendet `DigitalInput_CLK_I2` ein Ereignis über `IND` an den Ereigniseingang `EI2` von `RT_E_REND`.
   - Der Rücksetzeingang `R` von `RT_E_REND` wird über `DigitalInput_CLK_I3` (Tastendruck an `I3`) aktiviert.
   - Wenn beide Ereignisse an `EI1` und `EI2` innerhalb der Deadline (20 ms) eintreffen, erzeugt `RT_E_REND` ein Ereignis an seinem `EO`-Ausgang. Dieses Ereignis gelangt an den `CLK`-Eingang des Flip-Flops `E_T_FF`.
2. **Datenverkettung**:
   - Der Ausgang `Q` des Flip-Flops `E_T_FF` wird über eine Datenverbindung an den Dateneingang `OUT` des Ausgangsbausteins `DigitalOutput_Q1` übergeben.
3. **Zustandsänderung**:
   - Jedes erfolgreiche Rendezvous (gleichzeitiges Drücken von `I1` und `I2` innerhalb von 20 ms) toggelt den Ausgang `Q1`.
   - Ein Tastendruck an `I3` setzt den Rendezvous-Zustand zurück (ohne den Ausgang direkt zu ändern).
   - Der Ausgang `Q1` wechselt mit jedem Rendezvous-Ereignis seinen Wert.

Lernziele:

- Verständnis für Real-Time Rendezvous-Mechanismen in 4diac.
- Anwendung von Toggle-Flip-Flop und Ausgangsansteuerung.

- Erstellen von Ereignis- und Datenverbindungen über SubApp-Netzwerke.

Schwierigkeitsgrad: Fortgeschritten **Vorkenntnisse**: Grundlagen der Ereignissteuerung in 4diac, Umgang mit digitalen Ein-/Ausgängen.

Hinweise zur Ausführung: Die Übung wird auf einer Zielplattform mit logiBUS-Hardware ausgeführt. Die drei Taster müssen angeschlossen sein (Input_I1, Input_I2, Input_I3). Der Ausgang Output_Q1 kann z. B. eine LED ansteuern. Nach dem Start der Applikation ist das Flip-Flop zurückgesetzt (Q = FALSE).

Zusammenfassung

Die Übung zeigt, wie mit dem Baustein `RT_E_REND` ein echtzeitkritischer Rendezvous-Mechanismus realisiert wird. Durch die Kombination von digitalen Eingängen, einem Toggle-Flip-Flop und einem Ausgangsbaustein entsteht eine einfache aber praxisnahe Steuerung, bei der ein Ausgang nur dann umschaltet, wenn zwei Taster innerhalb einer kurzen Zeitspanne gleichzeitig gedrückt werden. Ein dritter Taster dient dem Zurücksetzen des Synchronisationszustands. Die Übung vertieft das Verständnis für zeitgesteuerte Ereignisverkettungen in IEC 61499.

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

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

2.2.1.49 Uebung_004a7: SR und T-Flip-Flop mit IE und E_REND (Rendezvous)



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_004a7`. Hier wird das Rendezvous-Muster mit einem erweiterten Flip-Flop-Typ kombiniert, der eine dedizierte Reset-Funktion besitzt.

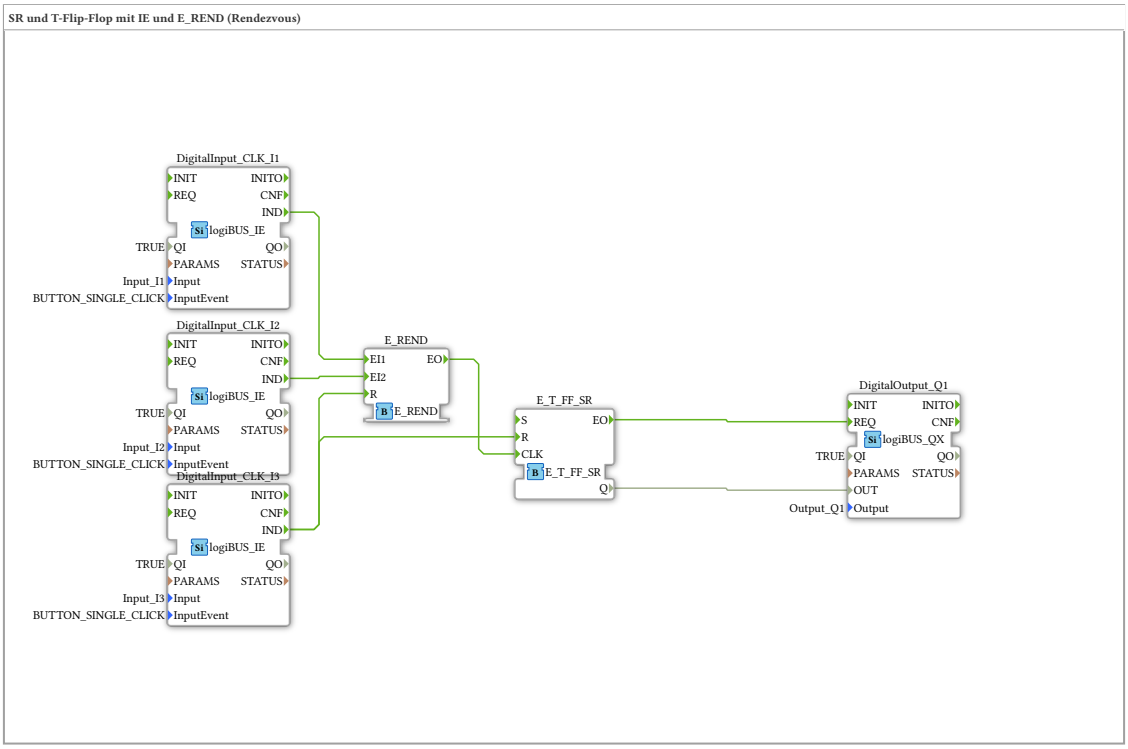
Ziel der Übung

Demonstration der Interaktion zwischen komplexer Ereignis-Logik (`E_REND`) und einem Flip-Flop mit Set/Reset-Priorität (`E_T_FF_SR`). Ziel ist eine Steuerung, die nur nach mehrfacher Bestätigung aktiv wird, aber jederzeit sicher abgeschaltet werden kann.

Beschreibung und Komponenten

Die Subapplikation `Uebung_004a7.SUB` nutzt drei Taster zur Steuerung eines Lampenzustands.

Funktionsbausteine (FBs)



- **`I1` & `I2`**: Eingänge für das Rendezvous (Scharfschalten).
- **`I3`**: Zentraler Reset-Eingang.
- **`E_REND`**: Synchronisiert die Ereignisse von `I1` und `I2`.
- **`E_T_FF_SR`**: Ein Toggle-Flip-Flop, das zusätzlich einen `R` (Reset) Eingang besitzt, um den Zustand definiert auf `FALSE` zu setzen.

Funktionsweise

```
<EventConnections>
    <Connection Source="DigitalInput_CLK_I1.IND" Destination="E_REND.EI1"/>
    <Connection Source="DigitalInput_CLK_I2.IND" Destination="E_REND.EI2"/>
    <Connection Source="E_REND.EO" Destination="E_T_FF.CLK"/>
    <Connection Source="DigitalInput_CLK_I3.IND" Destination="E_REND.R"/>
    <Connection Source="DigitalInput_CLK_I3.IND" Destination="E_T_FF.R"/>
</EventConnections>
```

1. Um das Licht (`Q1`) umzuschalten, müssen beide Taster `I1` und `I2` betätigt worden sein. Das Rendezvous feuert dann den Takt (`CLK`) für das Flip-Flop.
2. Der Taster `I3` fungiert als **Alles-Aus-Taste**:
   - Er setzt das Flip-Flop `E_T_FF_SR` sofort zurück (Ausgang wird `FALSE`).
   - Er löscht gleichzeitig das Gedächtnis von `E_REND`. Falls also nur ein Taster (`I1` oder `I2`) gedrückt war, wird diese Teil-Information gelöscht.

Anwendungsbeispiel

Maschinenfreigabe mit Not-Stopp: Zwei Sicherheitsbereiche müssen als „geprüft“ gemeldet werden (`I1` und `I2`), damit die Maschine in den nächsten Modus wechseln kann. Ein Not-Halt-Taster (`I3`) stoppt die Maschine jedoch sofort und macht alle bisherigen Sicherheits-Bestätigungen ungültig, sodass nach der Entriegelung beide Bereiche erneut geprüft werden müssen.

2.2.1.50 Uebung_004a8: mit E_SPLIT_2



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_004a8`. Dies ist eine Variante der Übung 004a4, bei der ein spezifischer Baustein für zwei Ausgänge verwendet wird.

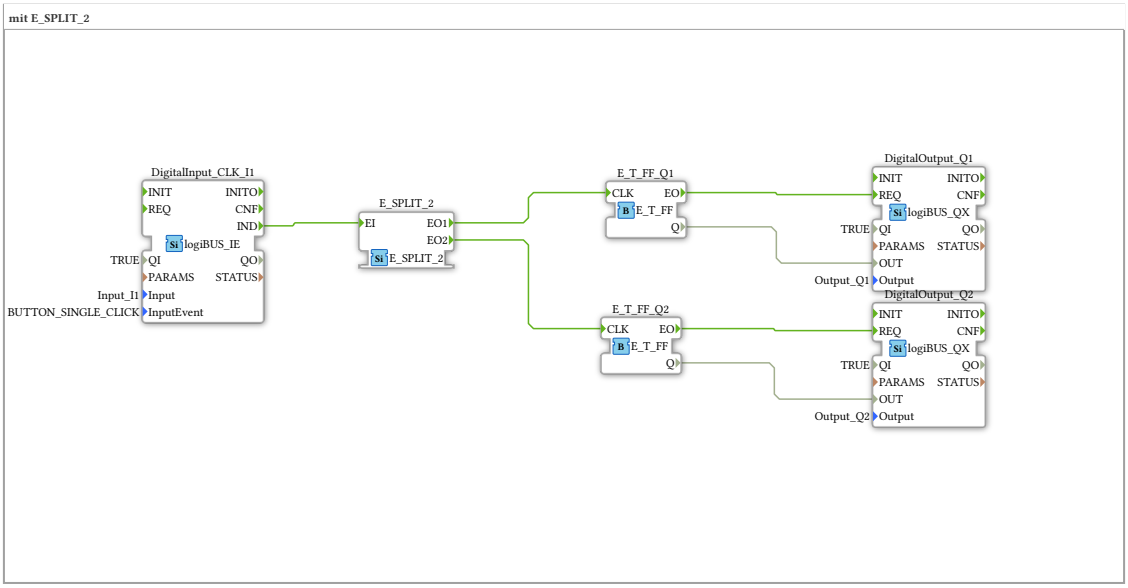
Ziel der Übung

Kennenlernen von typspezifischen Splitter-Bausteinen. Während `E_SPLIT` oft generisch ist, definieren Bausteine wie `E_SPLIT_2` explizit die Anzahl der Ausgänge.

Beschreibung und Komponenten

Die Subapplikation `Uebung_004a8.SUB` nutzt einen `E_SPLIT_2` Baustein zur Ereignisverteilung.

Funktionsbausteine (FBs)



- **`DigitalInput_CLK_I1`**: Taster.
- **`E_SPLIT_2`**: Verteilt den Eingang `EI` nacheinander auf `EO1` und `EO2`.
- **`E_T_FF_Q1` & `Q2`**: Zwei Flip-Flops.

Funktionsweise

Funktional identisch zu Übung 004a4: Ein einzelner Tastendruck führt dazu, dass zwei unabhängige Flip-Flops nacheinander getriggert werden. Dies stellt sicher, dass beide Speicherzustände sicher aktualisiert werden.

Anwendungsbeispiel

Synchrones Schalten von redundanten Systemen, bei denen sichergestellt sein muss, dass beide Teilsysteme den gleichen Schaltbefehl in einer festen Abfolge erhalten.

2.2.1.51 Uebung_004a9: mit E_SPLIT_3



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_004a9`. Hier wird das Konzept des sequenziellen Event-Splittings auf drei Ziele erweitert.

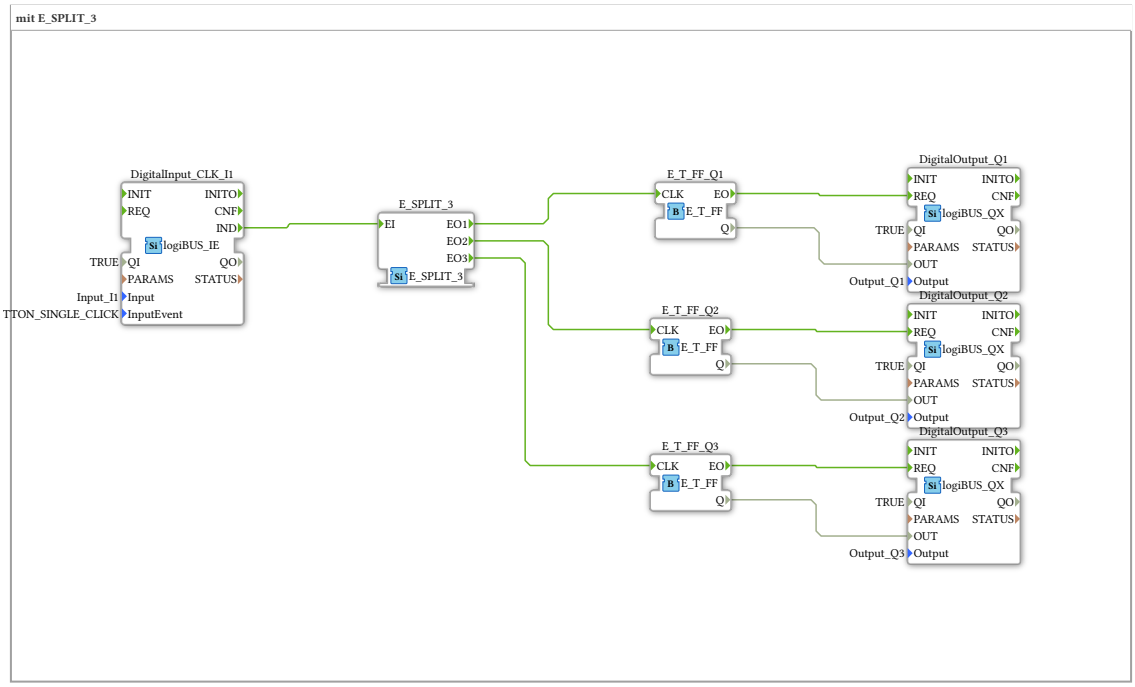
Ziel der Übung

Demonstration der Skalierbarkeit von Ereignis-Verteilern. Mit `E_SPLIT_3` können drei Prozesse mit einem einzigen Auslöser sequenziell angestoßen werden.

Beschreibung und Komponenten

Die Subapplikation `Uebung_004a9.SUB` verteilt das Signal eines Tasters auf drei separate Toggle-Flip-Flops und somit auf drei Ausgänge.

Funktionsbausteine (FBs)



- **`DigitalInput_CLK_I1`**: Der zentrale Auslöser (Taster).
- **`E_SPLIT_3`**: Verteilt den Eingang `EI` nacheinander auf `EO1`, `EO2` und `EO3`.
- **`E_T_FF_Q1, Q2, Q3`**: Drei unabhängige Flip-Flops.
- **`DigitalOutput_Q1, Q2, Q3`**: Drei physische Lampen.

Funktionsweise

Ein einziger Klick auf den Taster löst eine definierte Ereigniskette aus:

1. `EO1` feuert ➡️ `Q1` toggelt.
2. `EO2` feuert ➡️ `Q2` toggelt.
3. `EO3` feuert ➡️ `Q3` toggelt.

Die Abarbeitung erfolgt in der Steuerung so schnell, dass die Lampen für den Betrachter gleichzeitig umschalten, jedoch ist die interne Reihenfolge strikt vorgegeben.

Anwendungsbeispiel

Szenen-Schaltung im Gebäude: Ein Taster an der Wohnungstür schaltet gleichzeitig die Beleuchtung im Flur (`Q1`), in der Küche (`Q2`) und im Außenbereich (`Q3`) um. Durch den Splitter wird sichergestellt, dass alle Funktionsblöcke den Trigger erhalten.

2.2.1.52 Uebung_004b: Toggle Flip-Flop mit IE / E_SWITCH + E_SR

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_004b`. Hier wird gezeigt, wie die Funktion eines Toggle-Flip-Flops manuell aus Grundbausteinen (Weiche und Speicher) aufgebaut werden kann.

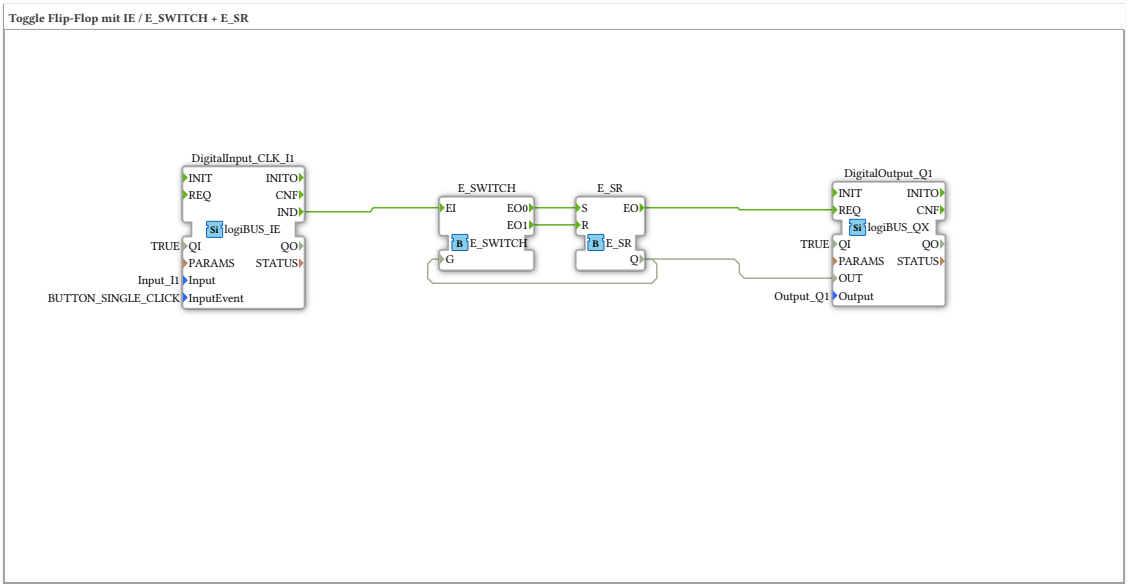
Ziel der Übung

Verständnis der inneren Logik eines Speicherbausteins. Anstatt den fertigen `E_T_FF` Baustein zu nutzen, wird hier eine Rückkopplungsschleife konstruiert, die den aktuellen Zustand nutzt, um das nächste Ereignis an den richtigen Eingang (`Setzen` oder `Rücksetzen`) zu leiten.

Beschreibung und Komponenten

Die Subapplikation `Uebung_004b.SUB` realisiert die Toggle-Funktion durch die Kombination einer Ereignis-Weiche und eines SR-Speichers.

Funktionsbausteine (FBs)



- **`DigitalInput_CLK_I1`**: Liefert ein Ereignis bei jedem Tastendruck.
- **`E_SWITCH`**: Eine Ereignis-Weiche. Je nach Zustand am Dateneingang `G` leitet sie das Ereignis `EI` entweder an `EO0` (wenn FALSE) oder an `EO1` (wenn TRUE) weiter.
- **`E_SR`**: Ein ereignisbasierter SR-Speicher (Set/Reset).
- **`DigitalOutput_Q1`**: Der Hardware-Ausgang.

Funktionsweise

Der Schlüssel liegt in der Rückführung des Ausgangszustands zum Eingang der Weiche:

```
<EventConnections>
    <Connection Source="DigitalInput_CLK_I1.IND" Destination="E_SWITCH.EI"/>
    <Connection Source="E_SWITCH.EO0" Destination="E_SR.S"/>
    <Connection Source="E_SWITCH.EO1" Destination="E_SR.R"/>
    <Connection Source="E_SR.EO" Destination="DigitalOutput_Q1.REQ"/>
</EventConnections>
<DataConnections>
    <Connection Source="E_SR.Q" Destination="DigitalOutput_Q1.OUT"/>
    <Connection Source="E_SR.Q" Destination="E_SWITCH.G"/>
</DataConnections>
```

Der funktionale Ablauf:

1. **Zustand AUS**: `E_SR.Q` ist FALSE, damit ist auch `E_SWITCH.G` auf FALSE.
2. Ein Tastendruck feuert `EI`. Die Weiche leitet es an `EO0` ➡️ `E_SR.S`. Der Speicher wird gesetzt, die Lampe geht an.
3. **Zustand AN**: Da die Lampe nun an ist, liegt an `E_SWITCH.G` eine TRUE an.
4. Der nächste Tastendruck feuert wieder `EI`. Diesmal leitet die Weiche das Event an `EO1` ➡️ `E_SR.R`. Der Speicher wird zurückgesetzt, die Lampe geht aus.

Bewertung

Dieser Aufbau ist lehrreich, um die Konzepte von Ereignissteuerung und Datenrückkopplung zu verstehen. In realen Projekten sollte jedoch immer der spezialisierte Baustein `E_T_FF` bevorzugt werden, da dieser übersichtlicher ist und weniger Ressourcen verbraucht.

2.2.1.53 Uebung_004b2: Toggle Flip-Flop mit IE / Split / doppelt



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_004b2`. Hier wird die manuelle Toggle-Logik aus Übung 004b auf zwei unabhängige Kanäle erweitert.

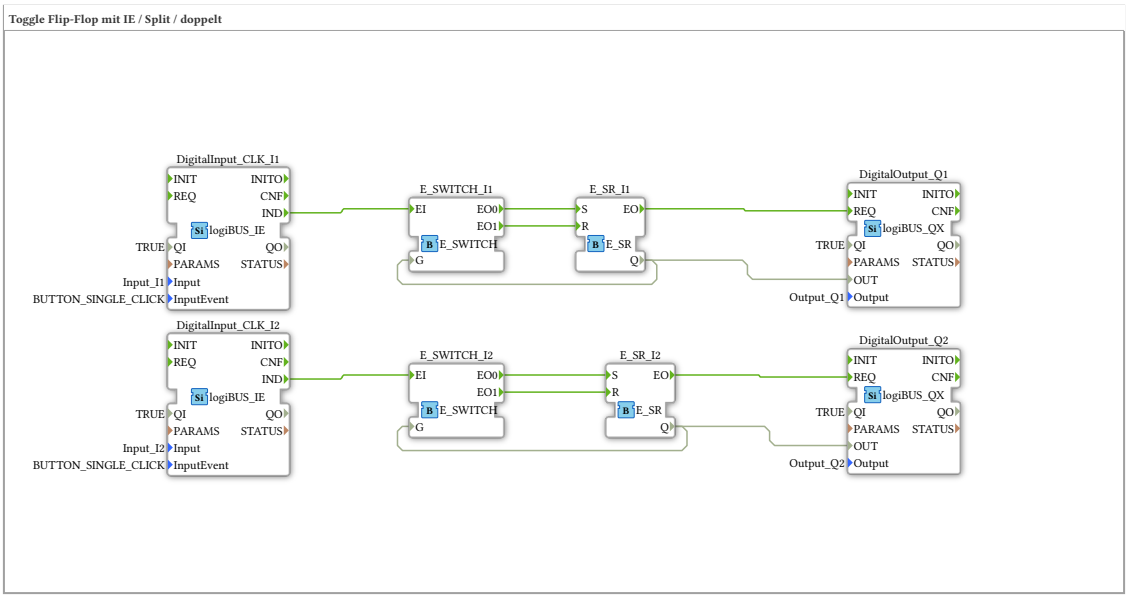
Ziel der Übung

Vertiefung des Verständnisses für parallele, rückgekoppelte Logikstrukturen. Jeder Kanal muss seinen eigenen Zustand korrekt verwalten, um unabhängig schaltbar zu sein.

Beschreibung und Komponenten

In `Uebung_004b2.SUB` sind zwei identische Logik-Stränge (Switch + Speicher) nebeneinander aufgebaut.

Funktionsbausteine (FBs)



- **Kanal 1**: Taster `I1`, Weiche `E_SWITCH_I1`, Speicher `E_SR_I1`, Ausgang `Q1`.
- **Kanal 2**: Taster `I2`, Weiche `E_SWITCH_I2`, Speicher `E_SR_I2`, Ausgang `Q2`.

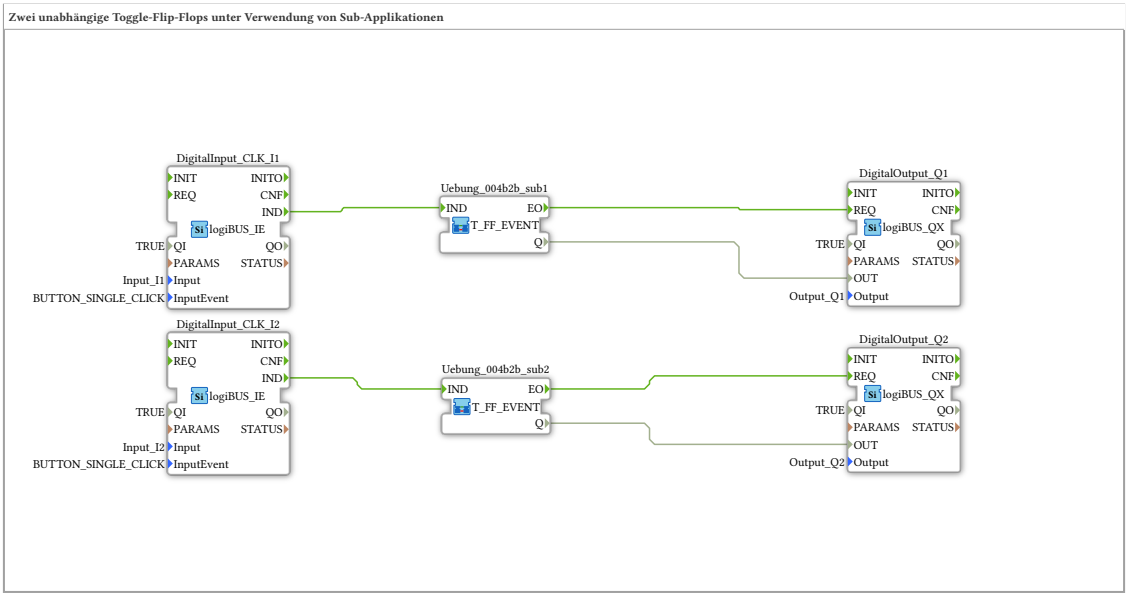
Funktionsweise

Die beiden Kanäle arbeiten nach dem gleichen Prinzip wie in Übung 004b: Der Ausgangszustand (`Q`) steuert über den Gate-Eingang (`G`) der Weiche, ob der nächste Tastendruck ein Setz- oder Rücksetz-Ereignis auslöst.

```
<!-- Beispiel Kanal 1 -->
<Connection Source="DigitalInput_CLK_I1.IND" Destination="E_SWITCH_I1.EI"/>
<Connection Source="E_SWITCH_I1.EO0" Destination="E_SR_I1.S"/>
<Connection Source="E_SWITCH_I1.EO1" Destination="E_SR_I1.R"/>
<Connection Source="E_SR_I1.Q" Destination="E_SWITCH_I1.G"/>
```

Da keine Querverbindungen zwischen den Strängen existieren, beeinflusst die Bedienung von Taster 1 niemals den Zustand von Lampe 2 und umgekehrt.

2.2.1.54 Uebung_004b2b: Zwei unabhängige Toggle-Flip-Flops unter Verwendung von Sub-Applikationen



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung realisiert zwei voneinander unabhängige Toggle-Flip-Flops.
Die Logik wird in einer wiederverwendbaren Sub-Applikation gekapselt, die zweimal instanziiert wird.
Als Eingabe dienen zwei Taster (Singelklick), als Ausgabe zwei digitale Ausgänge.
Bei jedem Tastendruck wechselt der zugehörige Ausgang seinen Zustand (Ein/Aus).

Verwendete Funktionsbausteine (FBs)

Haupt-FBs auf oberster Ebene

- **logiBUS_IE** – Ereignisgesteuerter Digitaleingang
  - Parameter: `Input` = `Input_I1` bzw. `Input_I2`, `InputEvent` = `BUTTON_SINGLE_CLICK`
  - Empfängt ein Ereignis, sobald der angeschlossene Taster einfach gedrückt wird.
- **logiBUS_QX** – Digitalausgang
  - Parameter: `Output` = `Output_Q1` bzw. `Output_Q2`
  - Setzt den physischen Ausgang entsprechend dem anliegenden Datenwert (BOOL).

Sub-Bausteine: `Uebung_004b2b_sub`

- **Typ**: SubAppType
- **Beschreibung**: Sub-Applikation für ein Toggle-Flip-Flop (besteht aus `E_SWITCH` und `E_SR`)

Verwendete interne FBs

- **`E_SWITCH_I1`**: Typ `E_SWITCH`
  - Ereigniseingang: `EI`
  - Ereignisausgänge: `EO0` (bei G=FALSE), `EO1` (bei G=TRUE)
  - Dateneingang: `G` (BOOL) – entscheidet über die Ereignisweiterleitung
- **`E_SR_I1`**: Typ `E_SR` (Set-Reset-Flip-Flop)
  - Ereigniseingänge: `S` (Set), `R` (Reset)
  - Ereignisausgang: `EO`
  - Datenausgang: `Q` (BOOL) – aktueller Zustand

Funktionsweise

1. Die Sub-Applikation empfängt ein Ereignis am Eingang `IND`.

2. Der interne E_SWITCH prüft den Wert seines Dateneingangs G (der mit dem aktuellen Ausgang Q des E_SR verbunden ist):
   - Ist Q = FALSE (G=0), wird das Ereignis an EO0 und damit an den Set-Eingang (S) des E_SR weitergeleitet.
   - Ist Q = TRUE (G=1), wird das Ereignis an EO1 und damit an den Reset-Eingang (R) des E_SR weitergeleitet.
3. Der E_SR wechselt daraufhin seinen Zustand:
   - Bei einem Set-Ereignis wird Q = TRUE.
   - Bei einem Reset-Ereignis wird Q = FALSE.
4. Nach der Zustandsänderung wird ein Ereignis am Ausgang EO erzeugt und der neue Wert von Q über den Ausgang der Sub-Applikation bereitgestellt.

Dadurch ergibt sich ein **Toggle-Verhalten**: Jedes ankommende Ereignis ändert den Ausgangszustand.

Programmablauf und Verbindungen

Die Haupt-SubApp Uebung_004b2b enthält zwei vollständig unabhängige Kanäle – je einen für die digitale Eingabe I1/I2 und Ausgabe Q1/Q2.

Verschaltung je Kanal:

```
logiBUS_IE (Taster) --> IND der Sub-Applikation
Sub-Applikation.EO --> REQ des logiBUS_QX
Sub-Applikation.Q   --> OUT des logiBUS_QX
```

- **Ereignisverbindungen**:
  - DigitalInput_CLK_I1.IND ⟶ Uebung_004b2b_sub1.IND
  - Uebung_004b2b_sub1.EO ⟶ DigitalOutput_Q1.REQ
  - (Analog für den zweiten Kanal mit I2 und Q2)
- **Datenverbindungen**:
  - Uebung_004b2b_sub1.Q ⟶ DigitalOutput_Q1.OUT
  - (Analog für den zweiten Kanal)

Durch diese Struktur wird die Logik der Sub-Applikation zweimal genutzt, ohne sie neu definieren zu müssen. Die Toggle-Funktion wird bei jedem Tastendruck (Singelklick) ausgeführt.

Zusammenfassung

- **Lernziele**:
  - Aufbau und Verwendung von Sub-Applikationen zur Wiederverwendung von Logik
  - Realisierung eines Toggle-Flip-Flops mit E_SWITCH und E_SR
  - Verkettung von ereignisgesteuerten und datengetriebenen Verbindungen
  - Parametrierung von hardwarenahen Ein-/Ausgangsbausteinen (logiBUS)
- **Schwierigkeitsgrad**: Fortgeschrittene Grundlagen
- **Vorkenntnisse**: Umgang mit Ereignissen, Boolesche Logik, einfache Flip-Flops

Die Übung demonstriert, wie modulare, wiederverwendbare Funktionsbausteine in einer industriellen Steuerungsumgebung (IEC 61499) effizient eingesetzt werden können.

2.2.1.55 Uebung_004b2b_sub

Hinweis

Diese Übung existiert nicht mehr als eigene Übung. Der Baustein wurde in die Bibliothek verschoben und heißt jetzt T_FF_EVENT. Beschreibung, Schnittstelle und Netzwerk stehen dort.

2.2.1.56 Uebung_004b3: Toggle Flip-Flop mit IE / Split / Verriegelt



Dieser Artikel beschreibt die logiBUS®-Übung Uebung_004b3. Diese Übung erweitert das zweikanalige System um eine gegenseitige Verriegelung: Es kann immer nur maximal eine Lampe gleichzeitig leuchten.

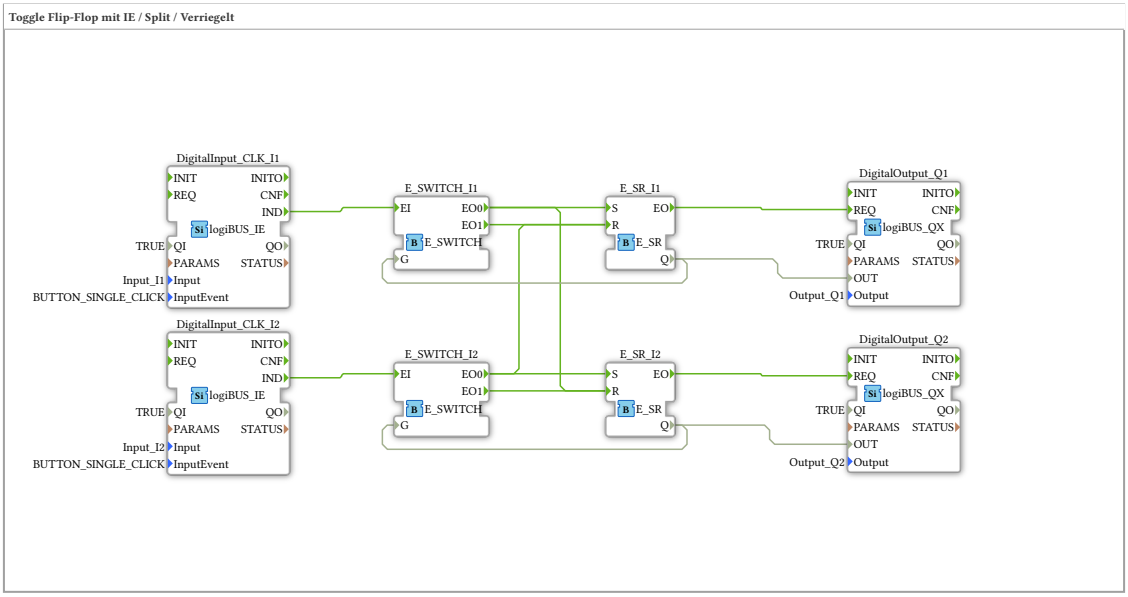
Ziel der Übung

Implementierung einer exklusiven Auswahl-Logik. Das Einschalten eines Kanals muss zwangsläufig das Ausschalten des anderen Kanals zur Folge haben. Dies ist eine Standardanforderung bei der Auswahl von Betriebsmodi oder Fahrtrichtungen.

Beschreibung und Komponenten

Die Subapplikation `Uebung_004b3.SUB` basiert auf dem Aufbau von 004b2, führt jedoch zusätzliche Ereignisverbindungen zur Verriegelung ein.

Funktionsbausteine (FBs)



- Identisch zu 004b2: Taster `I1/I2`, Weichen `E_SWITCH_I1/I2`, Speicher `E_SR_I1/I2`.

Funktionsweise

Die Besonderheit liegt in der „Über-Kreuz-Verbindung“ der Setz-Ereignisse:

```
<EventConnections>
    <!-- Normale Toggle-Logik Kanal 1 -->
    <Connection Source="E_SWITCH_I1.EO0" Destination="E_SR_I1.S"/>
    <Connection Source="E_SWITCH_I1.EO1" Destination="E_SR_I1.R"/>

    <!-- Verriegelung: Wenn Kanal 1 einschaltet (EO0), schalte Kanal 2 aus! -->
    <Connection Source="E_SWITCH_I1.EO0" Destination="E_SR_I2.R"/>

    <!-- Verriegelung: Wenn Kanal 2 einschaltet (EO0), schalte Kanal 1 aus! -->
    <Connection Source="E_SWITCH_I2.EO0" Destination="E_SR_I1.R"/>
</EventConnections>
```

Der funktionale Ablauf:

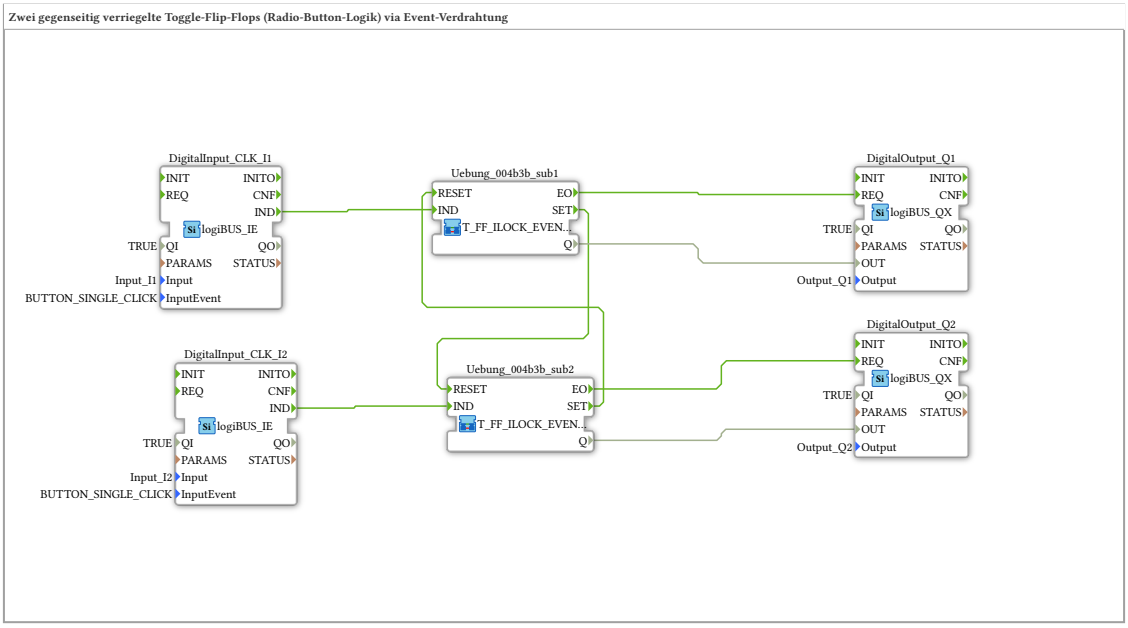
1. Lampe 1 ist an, Lampe 2 ist aus.
2. Nutzer drückt Taster 2 (`I2`).
3. Die Weiche von Kanal 2 erkennt „Lampe 2 ist aus“ und feuert das Ereignis zum Einschalten (`EO0`).
4. Dieses Ereignis geht einerseits an den Speicher von Kanal 2 (`Setzen`) ➡️ Lampe 2 geht an.
5. Gleichzeitig geht das selbe Ereignis an den Reset-Eingang von Kanal 1 ➡️ Lampe 1 geht sofort aus.

Ergebnis: Durch das Einschalten einer Funktion wird die andere automatisch deaktiviert.

Anwendungsbeispiel

Betriebsarten-Wahl: Eine Anlage kann entweder im Modus „Automatik“ (`Q1`) oder im Modus „Handbetrieb“ (`Q2`) laufen. Sobald der Bediener auf Handbetrieb umschaltet, muss die Automatik aus Sicherheitsgründen sofort gestoppt werden.

2.2.1.57 Uebung_004b3b: Zwei gegenseitig verriegelte Toggle-Flip-Flops (Radio-Button-Logik) via Event-Verdrahtung



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung realisiert eine **Radio-Button-Logik** – zwei Toggle-Flip-Flops, die sich gegenseitig verriegeln. Durch die Ereignisverdrahtung wird sichergestellt, dass immer nur einer der beiden Ausgänge (`Q1` oder `Q2`) aktiv sein kann. Ein Tastendruck auf den zugehörigen Eingang setzt das entsprechende Flip-Flop und setzt das andere gleichzeitig zurück. Die Logik ist als wiederverwendbare SubApplikation aufgebaut und nutzt logiBUS-Ein-/Ausgänge.

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: `Uebung_004b3b_sub`

Die Hauptapplikation verwendet zwei Instanzen dieses Sub-Bausteins (`Uebung_004b3b_sub1` und `Uebung_004b3b_sub2`). Der Sub-Baustein bildet ein **Toggle-Flip-Flop mit externem RESET-Eingang und SET-Ausgang zur Verriegelung**.

- **Typ**: SubApplikation (eigener Typ `Uebung_004b3b_sub`)
- **Schnittstelle**:
  - Ereigniseingänge: `IND` (Toggle-Impuls), `RESET` (externer Rücksetzbefehl)
  - Ereignisausgänge: `EO` (Ausgangsimpuls nach Toggle), `SET` (Signal an das andere Flip-Flop zum Zurücksetzen)
  - Datenausgang: `Q` (BOOL – aktueller Zustand)
- **Verwendete interne FBs**:
  - **E_SWITCH_I1**: `iec61499::events::E_SWITCH`
    - Parameter: keine
    - Ereigniseingang: `EI`
    - Daten-Eingang: `G` (BOOL)
    - Ereignisausgänge: `EO0` (wenn `G = FALSE`), `EO1` (wenn `G = TRUE`)
    - **Funktion**: Leitet den eingehenden Ereignisimpuls abhängig vom Wert von `G` entweder auf `EO0` oder `EO1`.
  - **E_SR_I1**: `iec61499::events::E_SR`
    - Parameter: keine
    - Ereigniseingänge: `S` (Set), `R` (Reset)
    - Ereignisausgang: `EO` (wird bei jedem Set- oder Reset-Ereignis ausgelöst)

  - Datenausgang: `Q` (BOOL – gespeicherter Zustand)
  - **Funktion**: Set-Reset-Flip-Flop. Wird durch ein Ereignis auf `S` gesetzt (`Q = TRUE`), durch ein Ereignis auf `R` zurückgesetzt (`Q = FALSE`). Der Ausgang `EO` signalisiert eine Zustandsänderung.

- **Funktionsweise des Sub-Bausteins**:
  - Ein Ereignis auf `IND` wird durch `E_SWITCH` geleitet.
  - Der aktuelle Zustand `Q` des Flip-Flops wird als `G` an den Switch geführt.
    - Ist `Q = FALSE`, wird das Ereignis auf `EO0` (Set-Pfad) gegeben → setzt `E_SR`.
    - Ist `Q = TRUE`, wird das Ereignis auf `EO1` (Reset-Pfad) gegeben → setzt `E_SR` zurück (Toggle-Verhalten).
  - Gleichzeitig wird das Set-Ereignis (von `EO0`) auch auf den `SET`-Ausgang gelegt, um das andere Flip-Flop zu verriegeln.
  - Der externe `RESET`-Eingang zwingt das Flip-Flop in den Zustand `Q = FALSE`.

Programmablauf und Verbindungen

Die Verschaltung der Hauptapplikation (`Uebung_004b3b`) ist wie folgt aufgebaut:

1. **Eingänge**: Zwei logiBUS-Digitaleingänge (`DigitalInput_CLK_I1` und `DigitalInput_CLK_I2`) werden mit den physikalischen Eingängen `Input_I1` und `Input_I2` verbunden. Beide sind auf das Ereignis `BUTTON_SINGLE_CLICK` konfiguriert – ein Tastendruck löst ein Ereignis aus.
2. **Verbindung der Ereignisse**:
   - Das Ereignis `IND` von `DigitalInput_CLK_I1` wird an den Sub-Baustein `Uebung_004b3b_sub1.IND` geführt.
   - Das Ereignis `IND` von `DigitalInput_CLK_I2` wird an `Uebung_004b3b_sub2.IND` geführt.
   - Der `SET`-Ausgang von `sub1` ist mit dem `RESET`-Eingang von `sub2` verbunden.
   - Der `SET`-Ausgang von `sub2` ist mit dem `RESET`-Eingang von `sub1` verbunden. *Dies bewirkt die gegenseitige Verriegelung: Sobald ein Flip-Flop gesetzt wird, wird das andere zurückgesetzt.*
3. **Ausgänge**:
   - Der `Q`-Ausgang von `sub1` wird an den logiBUS-Digitalausgang `DigitalOutput_Q1` (`Output_Q1`) übergeben.
   - Der `Q`-Ausgang von `sub2` wird an `DigitalOutput_Q2` (`Output_Q2`) übergeben.
   - Die Ausgangsimpulse (`EO`) der Sub-Bausteine triggern die zugehörigen Ausgangsbausteine über die `REQ`-Eingänge.

Ablauf Beispiel:

- Drückt man die Taste an `I1`, wird `sub1` getoggelt: Ist `Q1` aus, wird es eingeschaltet; der `SET`-Ausgang setzt `sub2` zurück.
- Ist `Q1` bereits an, schaltet es aus, ohne das andere zu beeinflussen.
- Ein Tastendruck auf `I2` verhält sich analog.

Lernziele:

- Verständnis von Ereignisverdrahtung in IEC 61499
- Aufbau einer gegenseitigen Verriegelung (Radio-Button)
- Verwendung von Toggle-Flip-Flops mit externem Reset
- Arbeit mit logiBUS-Ein-/Ausgangsbausteinen

Schwierigkeitsgrad: Mittel **Vorkenntnisse**: Grundlagen der ereignisgesteuerten Logik, Umgang mit 4diac-IDE und logiBUS.

Zusammenfassung

Die Übung `Uebung_004b3b` demonstriert eine elegante Realisierung einer Radio-Button-Logik durch zwei miteinander verriegelte Toggle-Flip-Flops. Die Kernidee besteht darin, die `SET`-Ausgänge der Sub-Bausteine kreuzweise mit den `RESET`-Eingängen zu verbinden, sodass nur ein Ausgang gleichzeitig aktiv sein kann. Dank der modularen Sub-Applikation lässt sich die Logik leicht auf mehrere Kanäle erweitern.

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

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

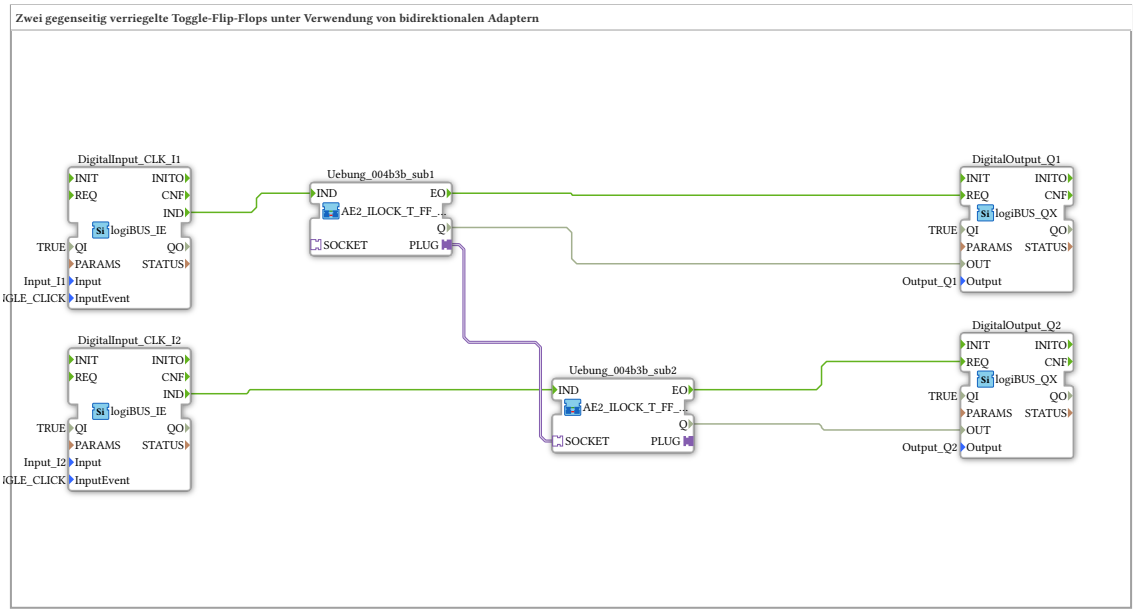
2.2.1.58 Uebung_004b3b_sub

Hinweis

Diese Übung existiert nicht mehr als eigene Übung. Der Baustein wurde in die Bibliothek verschoben und heißt jetzt T_FF_ILOCK_EVENT. Beschreibung, Schnittstelle und Netzwerk stehen dort.

2.2.1.59 Uebung_004b3c: Zwei gegenseitig verriegelte Toggle-Flip-Flops unter Verwendung von bidirektionalen Adaptern

(Der Übung liegt keine bildliche Darstellung bei.)



• – ○ • – ○ • – ○ •

Einleitung

In dieser Übung wird eine Schaltung aus zwei verriegelten Toggle-Flip-Flops realisiert. Jedes Flip-Flop wird über einen eigenen Taster (Eingang I1 bzw. I2) umgeschaltet. Die Besonderheit liegt in der gegenseitigen Verriegelung: Immer nur eines der beiden Flip-Flops kann den logischen Zustand `TRUE` annehmen. Sobald ein Flip-Flop auf `TRUE` gesetzt wird, setzt es automatisch das andere zurück. Die Kommunikation zwischen den beiden Sub-Applikationen erfolgt über einen einzigen bidirektionalen Adapter (Typ AE2), wodurch eine sehr kompakte Verbindungsstruktur entsteht.

Die Übung demonstriert den Einsatz von bidirektionalen Adaptern zur Signalübertragung zwischen Sub-Applikationen sowie die Kopplung von Ereignis- und Datenflüssen in einer verriegelten Steuerung.

Verwendete Funktionsbausteine (FBs)

Auf der obersten Ebene der Sub-Applikation `Uebung_004b3c` werden folgende Funktionsbausteine eingesetzt:

- **DigitalInput_CLK_I1** (Typ: `logiBUS::io::DI::logiBUS_IE`)
  - Eingang: `Input_I1`
  - Ereignis: `BUTTON_SINGLE_CLICK`
  - Gibt bei einem Tastendruck ein Ereignis an `IND` aus.
- **DigitalInput_CLK_I2** (Typ: `logiBUS::io::DI::logiBUS_IE`)
  - Eingang: `Input_I2`
  - Ereignis: `BUTTON_SINGLE_CLICK`
- **DigitalOutput_Q1** (Typ: `logiBUS::io::DQ::logiBUS_QX`)
  - Ausgang: `Output_Q1`
  - Schaltet den physikalischen Ausgang Q1.
- **DigitalOutput_Q2** (Typ: `logiBUS::io::DQ::logiBUS_QX`)

  - Ausgang: `Output_Q2`
- **Uebung_004b3b_sub1** (Typ: `Uebungen::Uebung_004b3c_sub`)
  - Erste Instanz des verriegelbaren Toggle-Flip-Flops.
- **Uebung_004b3b_sub2** (Typ: `Uebungen::Uebung_004b3c_sub`)
  - Zweite Instanz des verriegelbaren Toggle-Flip-Flops.

Sub-Bausteine: `Uebung_004b3c_sub`

Diese Sub-Applikation realisiert ein verriegelbares Toggle-Flip-Flop mit einer bidirektionalen Adapterschnittstelle (AE2).

- **Typ**: SubApp
- **Verwendete interne FBs**:
  - **E_SWITCH_I1** (Typ: `iec61499::events::E_SWITCH`)
    - Ereigniseingang: `EI`
    - Dateneingang: `G` (Gate)
    - Ereignisausgänge: `EO0` (bei G=FALSE), `EO1` (bei G=TRUE)
    - Funktion: Leitet das eingehende Ereignis abhängig vom Wert `G` entweder an `EO0` oder `EO1`.
  - **E_SR_I1** (Typ: `iec61499::events::E_SR`)
    - Ereigniseingänge: `S` (Set), `R` (Reset)
    - Ereignisausgang: `EO` (nach Änderung des Ausgangs Q)
    - Datenausgang: `Q` (BOOL)
    - Funktion: Set-Reset-Flip-Flop. Bei Set-Ereignis wird `Q = TRUE`, bei Reset-Ereignis `Q = FALSE`.
  - **AE2_EVENT_TO_E** (Typ: `adapter::conversion::bidirectional::AE2_EVENT_TO_E`)
    - Konvertiert ein Ereignis, das über den Adapter empfangen wird, in ein normales Ereignissignal.
  - **AE2_E_TO_EVENT** (Typ: `adapter::conversion::bidirectional::AE2_E_TO_EVENT`)
    - Konvertiert ein normales Ereignissignal in ein Ereignis, das über den Adapter gesendet wird.
- **Funktionsweise**: Der interne Toggle-Mechanismus wird durch das Set-Reset-Flip-Flop `E_SR` realisiert. Beim Eintreffen eines Ereignisses am Eingang `IND` wird durch `E_SWITCH` abhängig vom aktuellen Zustand `Q` (das auf `G` zurückgeführt ist) entschieden, ob gesetzt oder zurückgesetzt werden soll:
  - Ist `Q = FALSE` → Ereignis über `EO0` zum Set-Eingang `S` → Flip-Flop wird gesetzt.
  - Ist `Q = TRUE` → Ereignis über `EO1` zum Reset-Eingang `R` → Flip-Flop wird zurückgesetzt.

  Parallel dazu wird bei einem Setz-Vorgang (EO0) ein Ereignis über den Adapter (`AE2_E_TO_EVENT`) an die andere Sub-Applikation gesendet, um dort ein Reset auszulösen. Das über den Adapter empfangene Ereignis von der anderen Seite (`AE2_EVENT_TO_E`) wird ebenfalls auf den Reset-Eingang geführt. Dadurch wird sichergestellt, dass immer nur eines der beiden Flip-Flops `TRUE` ist.

  Der Ausgang `Q` wird nach außen über den Datenausgang der Sub-Applikation weitergegeben.

Programmablauf und Verbindungen

Die äußere Verschaltung der Haupt-Sub-Applikation ist wie folgt aufgebaut:

- Die Ereignisausgänge der beiden digitalen Eingänge (`DigitalInput_CLK_I1.IND` und `DigitalInput_CLK_I2.IND`) sind direkt mit den Ereigniseingängen `IND` der beiden Sub-Applikationen verbunden.
- Die Datenausgänge `Q` der Sub-Applikationen werden auf die digitalen Ausgangsbausteine `DigitalOutput_Q1` und `DigitalOutput_Q2` geführt.
- Der bidirektionale Adapter verbindet **Uebung_004b3b_sub1.PLUG** mit **Uebung_004b3b_sub2.SOCKET**. Diese einzige Verbindung reicht aus, um die gegenseitige Verriegelung zu realisieren: Immer wenn eine Sub-Applikation gesetzt wird, sendet sie ein Reset-Signal über den Adapter an die andere.

Lernziele der Übung:

- Verständnis des Einsatzes von bidirektionalen Adaptern zur Kopplung von Sub-Applikationen.
- Realisierung einer verriegelten Toggle-Flip-Flop-Struktur.
- Zusammenspiel von Ereignis- und Datenflüssen in IEC 61499.

- Praktischer Umgang mit Hardware-Ein-/Ausgängen (logiBUS).

Start der Übung: Die Übung kann direkt in der 4diac-IDE geöffnet und auf die Zielhardware (z. B. logiBUS) übertragen werden. Voraussetzung ist eine korrekte Konfiguration der Ein- und Ausgangsadressen entsprechend der Hardware.

Zusammenfassung

In dieser Übung wurde eine gegenseitig verriegelte Steuerung mit zwei Toggle-Flip-Flops realisiert. Die Verriegelung erfolgt über einen einzigen bidirektionalen Adapter, der die Reset-Signale zwischen den beiden Sub-Applikationen überträgt. Die Übung verdeutlicht, wie Adapter zur effizienten Kommunikation zwischen Sub-Applikationen genutzt werden können, und festigt das Verständnis von Ereignissteuerung, Zustandsspeicherung und Verriegelungslogik im IEC 61499-Modell.

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

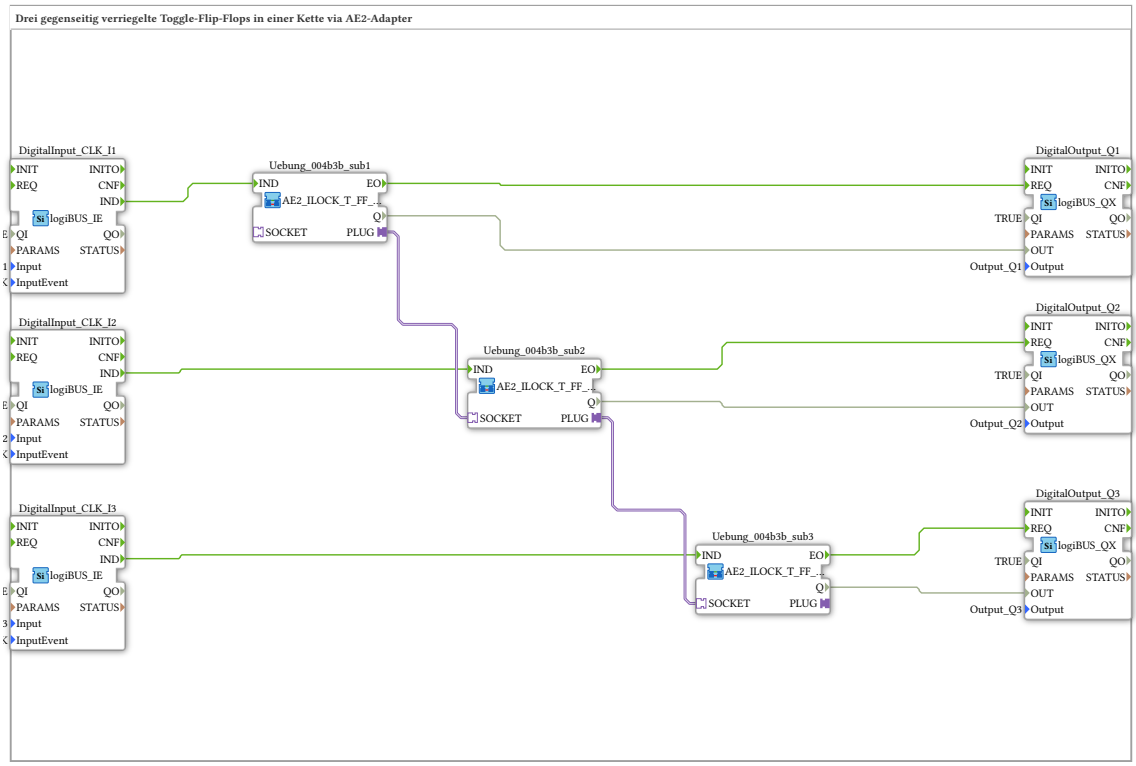
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.2.1.60 Uebung_004b3c_sub

Hinweis

Diese Übung existiert nicht mehr als eigene Übung. Der Baustein wurde in die Bibliothek verschoben und heißt jetzt AE2_ILOCK_T_FF_TO_BOOL_EVENT. Beschreibung, Schnittstelle und Netzwerk stehen dort.

2.2.1.61 Uebung_004b4c: Drei gegenseitig verriegelte Toggle-Flip-Flops in einer Kette via AE2-Adapter



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert eine Kette aus drei gegenseitig verriegelten Toggle-Flip-Flops. Jedes Flip-Flop wird über einen eigenen Taster (Single-Click) umgeschaltet. Die Verriegelung stellt sicher, dass immer nur genau ein Ausgang aktiv ist – ähnlich wie bei einem Ringzähler mit wechselseitiger Sperre. Die Verbindung zwischen den einzelnen Stufen erfolgt über einen bidirektionalen AE2-Adapter, sodass für die gesamte Kommunikation nur eine einzige Verbindung pro Kettenglied ausreicht.

Die Hardware besteht aus drei digitalen Eingängen (I1, I2, I3) und drei digitalen Ausgängen (Q1, Q2, Q3). Durch mehrmaliges Betätigen eines Tasters kann der aktive Ausgang weitergeschaltet werden.

Verwendete Funktionsbausteine (FBs)

- **logiBUS::io::DI::logiBUS_IE**: Digitaler Eingang mit Ereignisauslösung.
  - Parameter: QI = TRUE (Eingang aktiv), Input = entsprechender Hardware-Eingang (Input_I1, _I2, _I3), InputEvent = BUTTON_SINGLE_CLICK.
- **logiBUS::io::DQ::logiBUS_QX**: Digitaler Ausgang.
  - Parameter: QI = TRUE (Ausgang aktiv), Output = entsprechender Hardware-Ausgang (Output_Q1, _Q2, _Q3).
- **SubApp Uebung_004b3c_sub** (hier dreimal instanziiert): Kernbaustein der Übung. Er realisiert ein Toggle-Flip-Flop mit wechselseitiger Verriegelung.

Sub-Bausteine: Uebung_004b3c_sub

- **Typ**: SubApp (`Uebungen::Uebung_004b3c_sub`)
- **Interne Struktur** (angenommen, basierend auf dem Aufgabentext):
  - **Verwendete interne FBs** (nicht im XML sichtbar, aber konzeptionell):
    - Ein **Toggle-Flip-Flop** (z. B. ein SR- oder JK-Flipflop), das bei jedem Ereignis am Eingang `IND` seinen Zustand wechselt.
    - Ein **Verriegelungs-Logikblock**, der die Ausgänge der benachbarten Stufen über den Adapter (SOCKET/PLUG) auswertet und das eigene Flip-Flop zurücksetzt, wenn eine andere Stufe aktiv ist.
  - **Funktionsweise**:
    - Der Baustein hat einen Ereigniseingang `IND` und einen Dateneingang (vermutlich für die Verriegelung über den Adapter).
    - Bei einem positiven Ereignis (`IND`) wird der interne Zustand getoggelt. Gleichzeitig wird über den Adapter der Zustand der Nachbarstufe abgefragt. Nur wenn keine andere Stufe aktiv ist, darf der eigene Ausgang `Q` auf TRUE gehen; andernfalls wird er zurückgesetzt und gibt den Zustand an die nächste Stufe weiter.
    - Der Adapter (Plug/Socket) dient als bidirektionale Schnittstelle, sodass sowohl der Zustand der vorherigen als auch der nachfolgenden Stufe übertragen werden kann.

Programmablauf und Verbindungen

Die drei Bausteine sind hintereinandergeschaltet:

- **Ereignisverbindungen**:
  - Jeder Taster (`DigitalInput_CLK_I1`, `_I2`, `_I3`) löst beim Drücken ein Ereignis (`IND`) aus, das direkt an die entsprechende SubApp (`Uebung_004b3b_sub1`..`_sub3`) weitergeleitet wird.
  - Der Ausgang der SubApp (`EO`) triggert den zugehörigen digitalen Ausgangsbaustein (`DigitalOutput_Q1` usw.) über dessen `REQ`-Ereigniseingang.
- **Datenverbindungen**:
  - Der Ausgangswert `Q` jeder SubApp wird direkt mit dem Datenausgang `OUT` des zugehörigen `logiBUS_QX` verbunden, sodass der Hardware-Ausgang den aktuellen Zustand anzeigt.
- **Adapterverbindungen (Kette)**:
  - `Uebung_004b3b_sub1` (Stufe 1) verbindet seinen `PLUG` mit dem `SOCKET` von `Uebung_004b3b_sub2` (Stufe 2).
  - `Uebung_004b3b_sub2` verbindet seinen `PLUG` mit dem `SOCKET` von `Uebung_004b3b_sub3` (Stufe 3).
  - Dadurch entsteht eine logische Kette: Stufe 1 gibt ihren Zustand an Stufe 2 weiter, Stufe 2 an Stufe 3. Die bidirektionale Natur der Adapter ermöglicht es, dass sowohl Vorwärts- als auch Rückwärtssignale (z. B. Verriegelung) über ein einziges Kabelpaar ausgetauscht werden. Ein Kommentar im Netzwerk lautet: *„durch den Einsatz eines Bidirektionalen Adapters: 1 Verbindung REICHT!“*

Der Ablauf: Beim Drücken von Taster I1 wird Stufe 1 aktiviert (sofern keine andere aktiv ist). Der Zustand wandert dann bei weiteren Tastendrücken (I2 oder I3) entlang der Kette weiter. Nur ein Ausgang kann gleichzeitig TRUE sein – die Verriegelung verhindert mehrere aktive Stufen.

Zusammenfassung

Die Übung **Uebung_004b4c** veranschaulicht die Realisierung einer gegenseitig verriegelten Toggle-Flip-Flop-Kette unter Verwendung von bidirektionalen AE2-Adaptern. Sie kombiniert digitale Ein-/Ausgänge (logiBUS-Bibliothek) mit einer eigens entwickelten SubApp, die das Toggle- und Verriegelungsverhalten kapselt. Die Kopplung über Adapter reduziert die Anzahl der benötigten Verbindungen auf eine pro Stufe und vereinfacht so die Verkabelung. Die Übung eignet sich zum Erlernen des Adapter-Konzepts in 4diac sowie der Umsetzung von wechselseitiger Sperrlogik (Mutual Exclusion) in industriellen Steuerungen.

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

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

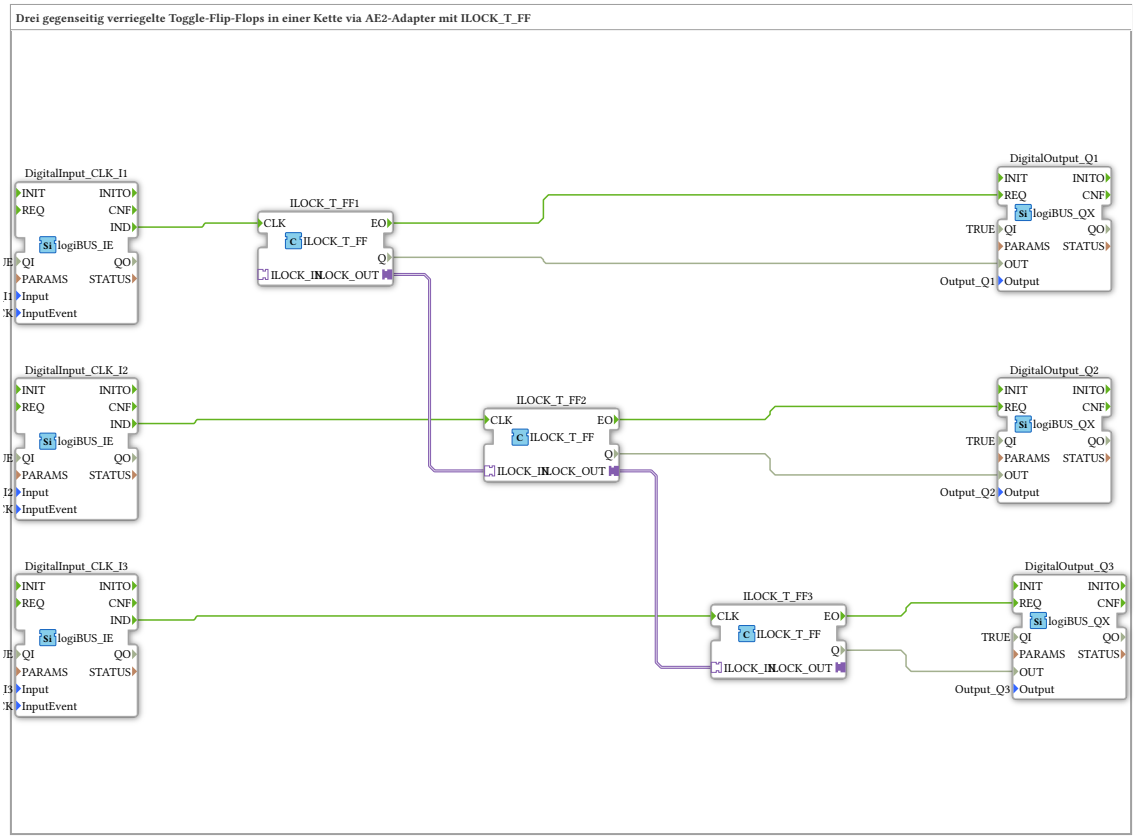
2.2.1.62 Uebung_004b4c_sub

Hinweis

Diese Übung existiert in den aktuellen Trainingsquellen nicht mehr (entfernt, zusammengelegt oder umbenannt). Diese Seite bleibt nur als Platzhalter für bestehende Verweise erhalten. Die aktuellen Übungen stehen in der Übungsübersicht.

2.2.1.63 Uebung_004b4d: Drei gegenseitig verriegelte Toggle-Flip-Flops in einer Kette via AE2-Adapter mit ILOCK_T_FF

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_004b4d (Drei gegenseitig verriegelte Toggle-Flip-Flops in einer Kette via AE2-Adapter mit ILOCK_T_FF).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Drei gegenseitig verriegelte Toggle-Flip-Flops in einer Kette via AE2-Adapter mit ILOCK_T_FF**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_004b4d.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QX.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
- **DigitalInput_CLK_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **DigitalOutput_Q2**: Instanz des Typs logiBUS::io::DQ::logiBUS_QX.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q2
- **DigitalInput_CLK_I2**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I2
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **ILOCK_T_FF1**: Instanz des Typs logiBUS::signalprocessing::interlock::ILOCK_T_FF.
- **ILOCK_T_FF2**: Instanz des Typs logiBUS::signalprocessing::interlock::ILOCK_T_FF.
- **ILOCK_T_FF3**: Instanz des Typs logiBUS::signalprocessing::interlock::ILOCK_T_FF.
- **DigitalOutput_Q3**: Instanz des Typs logiBUS::io::DQ::logiBUS_QX.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q3
- **DigitalInput_CLK_I3**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I3
  - Parameter InputEvent = BUTTON_SINGLE_CLICK

Verbindungen und Schnittstellen

Adapterverbindungen:

- ILOCK_T_FF1.ILOCK_OUT -> ILOCK_T_FF2.ILOCK_IN
- ILOCK_T_FF2.ILOCK_OUT -> ILOCK_T_FF3.ILOCK_IN

Ereignisverbindungen:

- DigitalInput_CLK_I1.IND -> ILOCK_T_FF1.CLK
- DigitalInput_CLK_I2.IND -> ILOCK_T_FF2.CLK
- DigitalInput_CLK_I3.IND -> ILOCK_T_FF3.CLK
- ILOCK_T_FF1.EO -> DigitalOutput_Q1.REQ
- ILOCK_T_FF2.EO -> DigitalOutput_Q2.REQ
- ILOCK_T_FF3.EO -> DigitalOutput_Q3.REQ

Datenverbindungen:

- ILOCK_T_FF1.Q -> DigitalOutput_Q1.OUT
- ILOCK_T_FF2.Q -> DigitalOutput_Q2.OUT
- ILOCK_T_FF3.Q -> DigitalOutput_Q3.OUT

Hinweise aus dem Modell

durch den Einsatz eines Bidirektionalen Adapters: 1 Verbindung REICHT !

Programmablauf und Funktionsweise

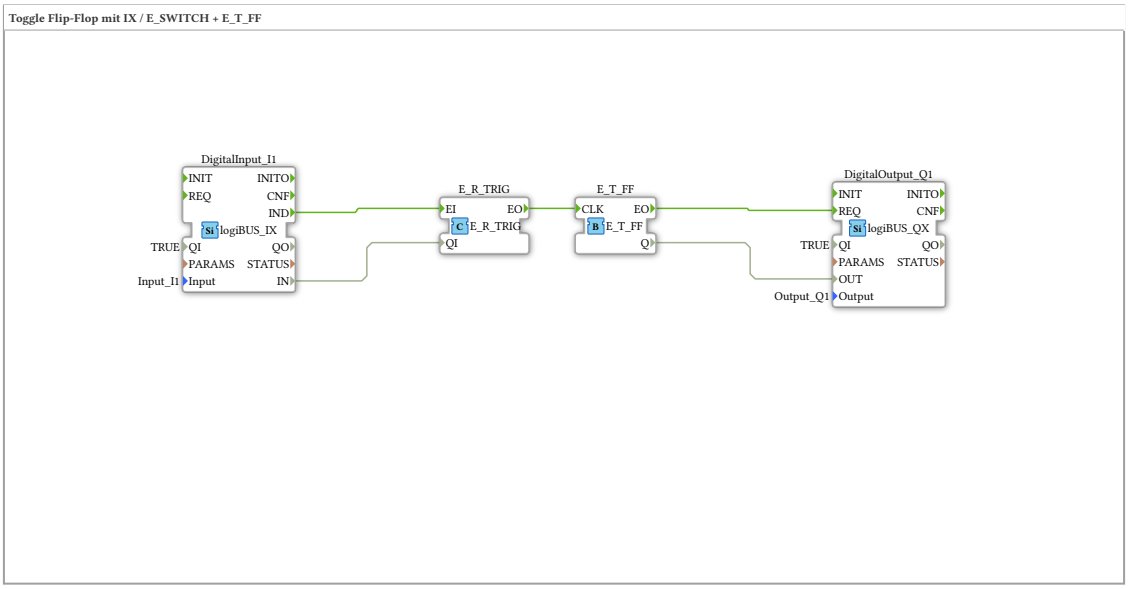
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.

3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_004b4d demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.2.1.64 Uebung_004b5: Toggle Flip-Flop mit IX / E_SWITCH + E_T_FF



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung demonstriert die Realisierung eines Toggle-Flipflops (T-FF) mithilfe der Funktionsbausteine `E_R_TRIG` (Flankenerkennung steigend) und `E_T_FF` (Toggle-Flipflop). Ein digitaler Eingang (IX) wird als Taster verwendet – jede steigende Flanke am Eingang schaltet den digitalen Ausgang (QX) um (toggle). Der Aufbau eignet sich z. B. zum Ein- und Ausschalten einer Leuchte mit einem einzelnen Taster.

Verwendete Funktionsbausteine (FBs)

Die Subapplikation besteht aus vier Funktionsbausteinen:

- **DigitalInput_I1**: Typ `logiBUS::io::DI::logiBUS_IX`
  - **Beschreibung**: Liest einen digitalen Eingang vom Feldbus (z. B. einen Taster).
  - **Parameter**: `QI = TRUE` (Qualität des Eingangs aktiv), `Input = Input_I1` (physikalischer Kanal).
  - **Ereignisausgänge**: `IND` (wird ausgelöst, wenn sich der Eingangszustand ändert).
  - **Datenausgang**: `IN` (aktueller digitaler Wert).
- **E_R_TRIG**: Typ `iec61499::events::E_R_TRIG`
  - **Beschreibung**: Erkennt eine steigende Flanke des Eingangssignals.
  - **Eingänge**:
    - Ereignis: `EI` (Start der Verarbeitung).
    - Daten: `QI` (Qualität des Eingangswerts, hier mit `DigitalInput_I1.IN` verbunden).
  - **Ausgänge**:
    - Ereignis: `EO` (wird genau bei einer steigenden Flanke von `QI` ausgelöst).
- **E_T_FF**: Typ `iec61499::events::E_T_FF`
  - **Beschreibung**: Toggle-Flipflop – bei jedem Ereignis am Eingang `CLK` wird der interne Zustand invertiert.
  - **Eingänge**:
    - Ereignis: `CLK` (Takt, hier von `E_R_TRIG.EO` gespeist).
  - **Ausgänge**:

  - Ereignis: `EO` (wird nach Zustandswechsel einmal ausgelöst).
  - Daten: `Q` (aktueller Zustand des Flipflops, TRUE/FALSE).

- **DigitalOutput_Q1**: Typ `logiBUS::io::DQ::logiBUS_QX`
  - **Beschreibung**: Setzt einen digitalen Ausgang am Feldbus (z. B. eine Leuchte).
  - **Parameter**: `QI = TRUE` (Ausgang freigegeben), `Output = Output_Q1` (physikalischer Kanal).
  - **Ereigniseingänge**: `REQ` (Anforderung zum Beschreiben des Ausgangs).
  - **Dateneingang**: `OUT` (gewünschter digitaler Wert).

Programmablauf und Verbindungen

Die Subapplikation ist als ereignisgesteuerte Kette realisiert:

1. **Eingangsänderung**: Der Baustein `DigitalInput_I1` überwacht den physikalischen Eingang. Sobald sich der Zustand ändert, wird das Ereignis `IND` ausgelöst.
2. **Flankenerkennung**: Das Ereignis `IND` wird an den Ereigniseingang `EI` von `E_R_TRIG` weitergeleitet (**Event-Verbindung**: `DigitalInput_I1.IND → E_R_TRIG.EI`). Parallel dazu wird der aktuelle digitale Wert (`DigitalInput_I1.IN`) an den Dateneingang `QI` von `E_R_TRIG` übergeben (**Data-Verbindung**: `DigitalInput_I1.IN → E_R_TRIG.QI`). `E_R_TRIG` prüft, ob der Wert von `QI` eine steigende Flanke (Wechsel von FALSE auf TRUE) aufweist. Ist dies der Fall, wird am Ausgang `EO` ein Ereignis erzeugt.
3. **Toggle-Flipflop**: Das Ereignis `EO` von `E_R_TRIG` triggert den Takteingang `CLK` von `E_T_FF` (**Event-Verbindung**: `E_R_TRIG.EO → E_T_FF.CLK`). Bei jedem Takt wechselt der Zustand des Flipflops. Das Ergebnis steht am Datenausgang `Q` zur Verfügung. Gleichzeitig wird das Ausgangsereignis `EO` von `E_T_FF` ausgelöst.
4. **Ausgang setzen**: Das Ereignis `EO` von `E_T_FF` wird an den `REQ`-Eingang von `DigitalOutput_Q1` weitergeleitet (**Event-Verbindung**: `E_T_FF.EO → DigitalOutput_Q1.REQ`). Der Flipflop-Zustand (`E_T_FF.Q`) wird als Sollwert an den Dateneingang `OUT` von `DigitalOutput_Q1` übergeben (**Data-Verbindung**: `E_T_FF.Q → DigitalOutput_Q1.OUT`). Dadurch wird der physikalische Ausgang entsprechend gesetzt.

Lernziele:

- Verständnis ereignisgesteuerter Abläufe in 4diac.
- Einsatz eines Flankenerkennungsbausteins (`E_R_TRIG`).
- Realisierung eines Toggle-Flipflops mit `E_T_FF`.
- Kopplung von digitalen Ein-/Ausgängen über logiBUS.

Schwierigkeitsgrad: Einsteiger (nach Kenntnis grundlegender FB-Typen).
Vorkenntnisse: Grundlegender Umgang mit 4diac-IDE, Kenntnis von Event- und Datenverbindungen.

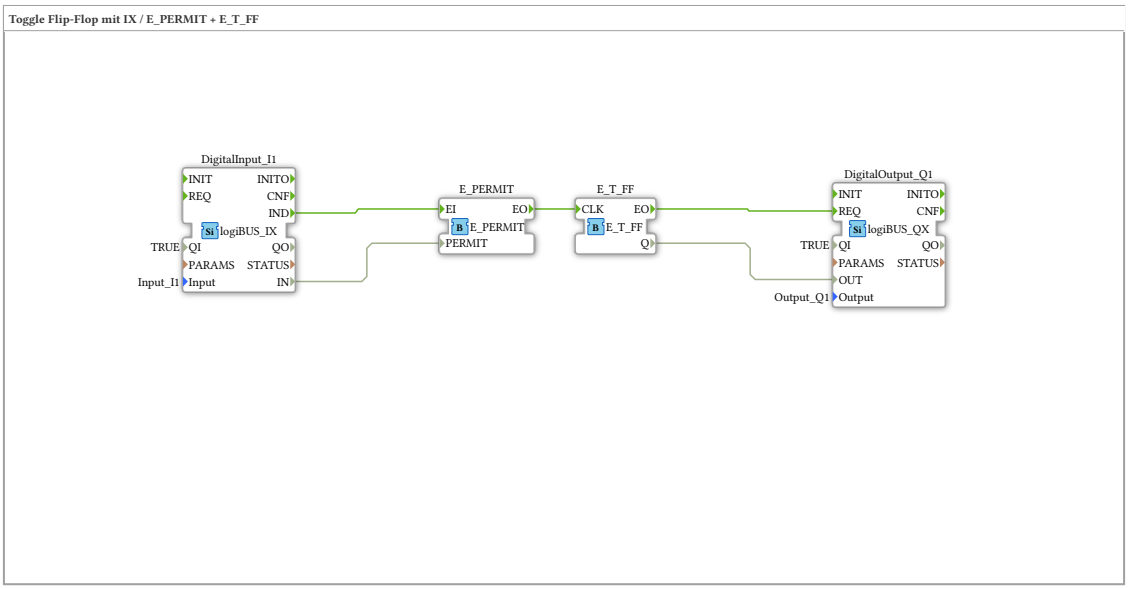
Zusammenfassung

Mit dieser Übung wurde eine typische Taster-Schalter-Funktion (Toggle) umgesetzt. Durch die Kombination von `E_R_TRIG` und `E_T_FF` wird jede steigende Flanke am Eingang erkannt und der Ausgangszustand umgeschaltet. Die Bausteine sind lose gekoppelt – ein Vorteil der ereignisgesteuerten Programmierung. Die Subapplikation kann direkt in ein 4diac-Projekt eingebunden und auf einem entsprechenden Zielsystem (mit logiBUS-Anbindung) ausgeführt werden.

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

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

2.2.1.65 Uebung_004b6: Toggle Flip-Flop mit IX / E_PERMIT + E_T_FF



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung realisiert einen Toggle-Flip-Flop (T-FF) mithilfe eines digitalen Eingangs (logiBUS_IX) und der Funktionsbausteine **E_PERMIT** und **E_T_FF**. Der digitale Eingang dient als Freigabesignal (PERMIT) für einen Takt, der den Zustand des Flip-Flops wechselt. Der Ausgang wird auf einen digitalen Ausgang (logiBUS_QX) geschaltet.

Verwendete Funktionsbausteine (FBs)

DigitalInput_I1

- **Typ**: `logiBUS::io::DI::logiBUS_IX`
- **Parameter**:
  - `QI` = `TRUE`
  - `Input` = `Input_I1`
- **Funktion**: Liest den Zustand des angeschlossenen digitalen Eingangssignals und stellt den Wert am Datenausgang `IN` bereit. Bei einer Signaländerung wird das Ereignis `IND` ausgelöst.

E_PERMIT

- **Typ**: `iec61499::events::E_PERMIT`
- **Parameter**: keine
- **Ereigniseingang/-ausgang**:
  - `EI` (Ereigniseingang)
  - `EO` (Ereignisausgang)
- **Dateneingang**:
  - `PERMIT` (BOOL)
- **Funktion**: Gibt ein an `EI` eingehendes Ereignis nur dann am Ausgang `EO` weiter, wenn der Wert von `PERMIT` = `TRUE` ist. Dient als Freigabeglied.

E_T_FF

- **Typ**: `iec61499::events::E_T_FF`
- **Parameter**: keine
- **Ereigniseingang/-ausgang**:
  - `CLK` (Ereigniseingang)
  - `EO` (Ereignisausgang)
- **Datenausgang**:
  - `Q` (BOOL)

- **Funktion**: Toggle-Flip-Flop. Bei jedem Takt (Ereignis an `CLK`) wechselt der Ausgang `Q` seinen Zustand (toggle). Gleichzeitig wird das Ereignis `EO` ausgelöst.

DigitalOutput_Q1

- **Typ**: `logiBUS::io::DQ::logiBUS_QX`
- **Parameter**:
  - `QI` = `TRUE`
  - `Output` = `Output_Q1`
- **Funktion**: Setzt den digitalen Ausgang gemäß dem am Datenausgang `OUT` anliegenden Wert. Die Aktualisierung erfolgt durch das Ereignis `REQ`.

Programmablauf und Verbindungen

Event-Verbindungen:

- `DigitalInput_I1.IND` ⟶ `E_PERMIT.EI`
- `E_PERMIT.EO` ⟶ `E_T_FF.CLK`
- `E_T_FF.EO` ⟶ `DigitalOutput_Q1.REQ`

Datenverbindungen:

- `DigitalInput_I1.IN` ⟶ `E_PERMIT.PERMIT`
- `E_T_FF.Q` ⟶ `DigitalOutput_Q1.OUT`

Ablauf: Sobald sich der Digitaleingang ändert, wird das Ereignis `IND` ausgelöst. Der aktuelle Zustand des Eingangs (`IN`) wird als `PERMIT` an den Baustein `E_PERMIT` übergeben.

- Ist `PERMIT = TRUE`, wird das Ereignis an den Takt-Eingang (`CLK`) des Toggle-Flip-Flops weitergeleitet.
- Der `E_T_FF` toggelt daraufhin seinen Ausgang `Q`.
- Gleichzeitig wird das Ereignis `EO` von `E_T_FF` an den Digitalausgang gesendet, der den neuen Wert von `Q` übernimmt.

Lernziele:

- Verständnis des Toggle-Flip-Flop (T-FF)
- Anwendung des Freigabebausteins `E_PERMIT`
- Zusammenspiel von Ereignis- und Datenflüssen in IEC 61499

Schwierigkeitsgrad: Mittel **Vorkenntnisse**: Grundlagen der ereignisgesteuerten Logik und digitaler Ein-/Ausgänge

Starten der Übung: Fügen Sie die SubApp `Uebung_004b6` in ein 4diac-Projekt ein, verbinden Sie die Hardware-Ressourcen (z. B. physischen Eingang `Input_I1` und Ausgang `Output_Q1`) und führen Sie die Applikation aus.

Zusammenfassung

Die Übung demonstriert einen Toggle-Flip-Flop, der nur bei aktivem digitalen Eingang seinen Zustand wechselt. Durch die Kombination von `E_PERMIT` und `E_T_FF` kann eine Freigabe für den Takt realisiert werden – nützlich für Anwendungen wie entprellte Taster oder Betriebsartenumschaltung.

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

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

2.2.1.66 Uebung_004c1: Toggle Flip-Flop mit IE mit BUTTON_DOUBLE_CLICK



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_004c1`. Ab hier widmen wir uns den erweiterten Fähigkeiten des `logiBUS_IE` Bausteins, der komplexe Taster-Muster erkennen kann.

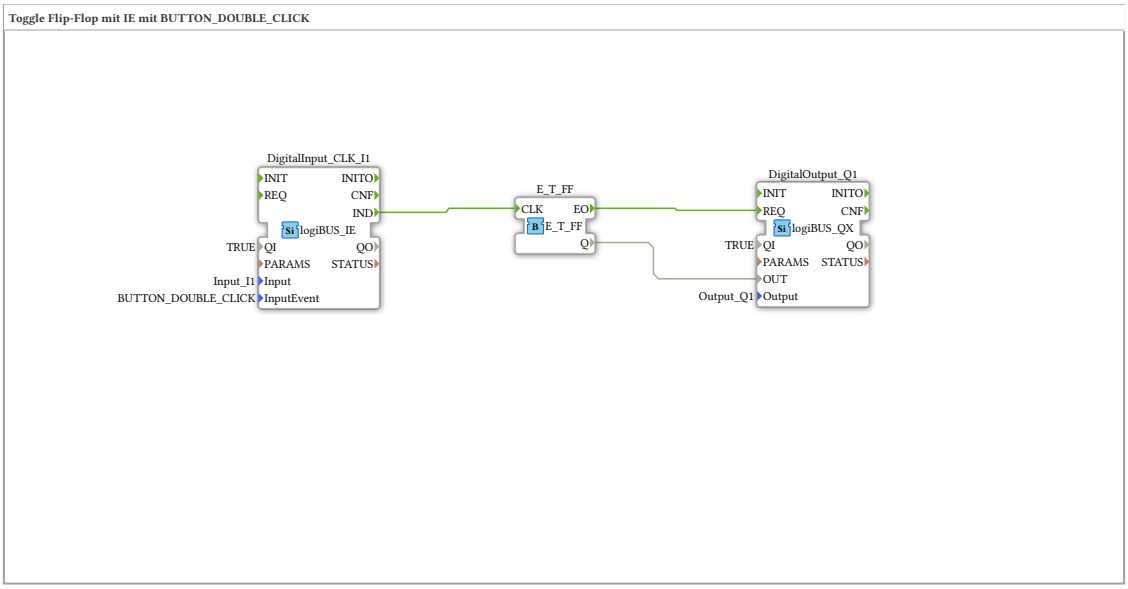
Ziel der Übung

Nutzung des Ereignisses `BUTTON_DOUBLE_CLICK` zur Steuerung einer Speicherfunktion.

Beschreibung und Komponenten

Die Subapplikation `Uebung_004c1.SUB` schaltet eine Lampe nur bei einem Doppelklick um.

Funktionsbausteine (FBs)



- **`DigitalInput_CLK_I1`**: Typ `logiBUS_IE`. Dieser ist im Parameter `InputEvent` auf `BUTTON_DOUBLE_CLICK` konfiguriert.
- **`E_T_FF`**: Das Toggle-Flip-Flop.

Funktionsweise

Der Eingangsbaustein überwacht das zeitliche Muster am Hardware-Pin `I1`.

1. Ein einfacher Tastendruck wird ignoriert (kein Event an `IND`).
2. Werden zwei Klicks innerhalb einer definierten Zeit (meist < 500ms) erkannt, feuert der Baustein **einmal** das Ereignis `IND`.
3. Dieses Ereignis triggert das Flip-Flop, welches den Zustand der Lampe wechselt.

Anwendungsbeispiel

Vermeidung von Fehlbedienungen: Kritische Befehle (wie z.B. „Alle Motoren Stopp“ oder „Daten löschen“) können auf einen Doppelklick gelegt werden, damit ein versehentliches Berühren des Tasters keine Folgen hat.

2.2.1.67 Uebung_004c2: Toggle Flip-Flop mit IE mit BUTTON_LONG_PRESS_START

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_004c2`.

Ziel der Übung

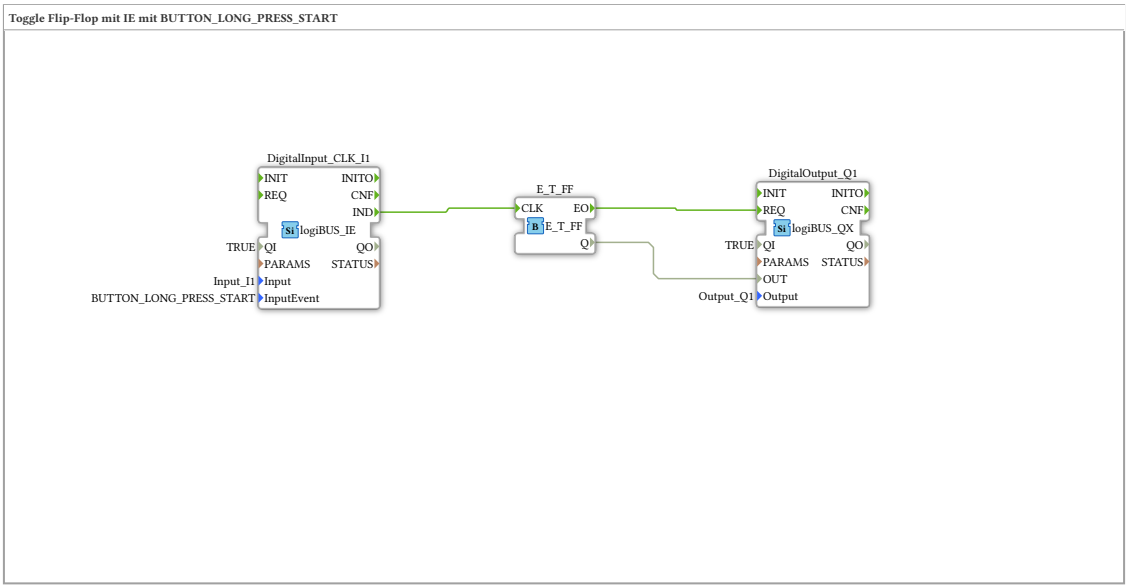
Nutzung des Ereignisses `BUTTON_LONG_PRESS_START`.

Funktionsweise

Der Baustein `DigitalInput_CLK_I1` in `Uebung_004c2.SUB` reagiert auf langes Drücken.

Das Ereignis `IND` wird genau in dem Moment gefeuert, in dem die vordefinierte Zeit für einen „langen Druck“ (z.B. 1 Sekunde) abgelaufen ist – auch wenn der Taster danach noch weiter gedrückt bleibt. Ein kurzes Tippen löst hier kein Ereignis aus.

Anwendungsbeispiel



Menü-Navigation: In vielen Steuerungen gelangt man durch einen kurzen Klick zur nächsten Seite, während ein langer Druck (`LONG_PRESS_START`) das Setup-Menü öffnet.

2.2.1.68 Uebung_004c3: Toggle Flip-Flop mit IE mit BUTTON_LONG_PRESS_UP



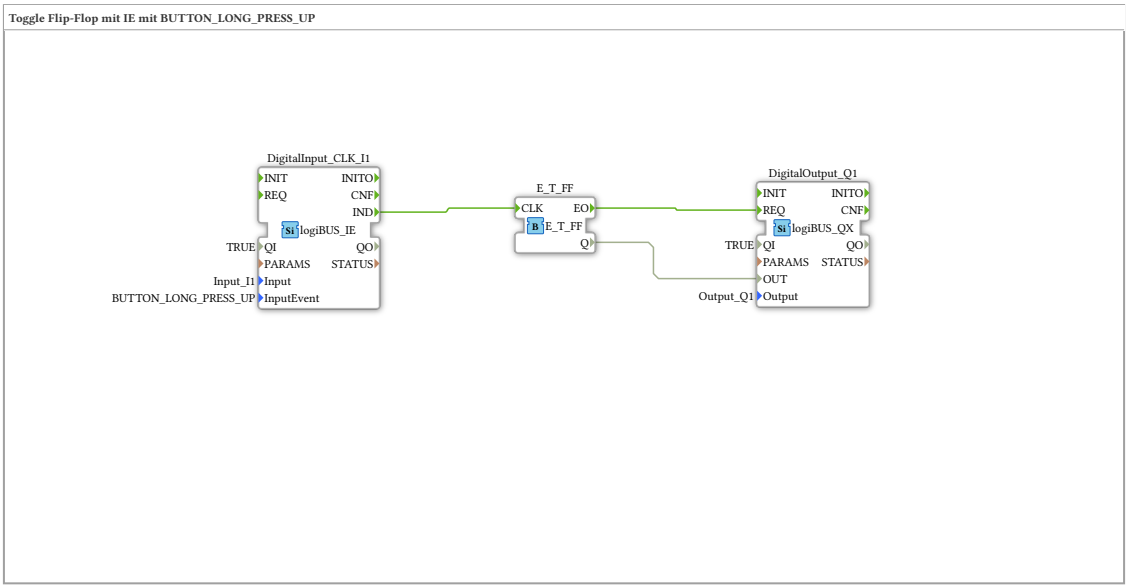
Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_004c3`.

Ziel der Übung

Nutzung des Ereignisses `BUTTON_LONG_PRESS_UP`.

Funktionsweise

Der Baustein `DigitalInput_CLK_I1` in `Uebung_004c3.SUB` erkennt das Ende eines langen Drucks.



Im Gegensatz zum `START`-Event feuert `LONG_PRESS_UP` erst dann, wenn der Nutzer den Taster **wieder loslässt**, sofern dieser vorher lange genug gedrückt wurde. Dies ermöglicht es, Aktionen genau am Ende einer Interaktion auszulösen.

Anwendungsbeispiel

Bestätigungs-Dialog: Der Nutzer muss eine Taste lange gedrückt halten, um sicherzugehen, dass er die Aktion will. Die Ausführung (z.B. „Motor starten“) erfolgt erst beim Loslassen als finale Bestätigung.

2.2.1.69 Uebung_004c4: Toggle Flip-Flop mit IE mit BUTTON_LONG_PRESS_HOLD



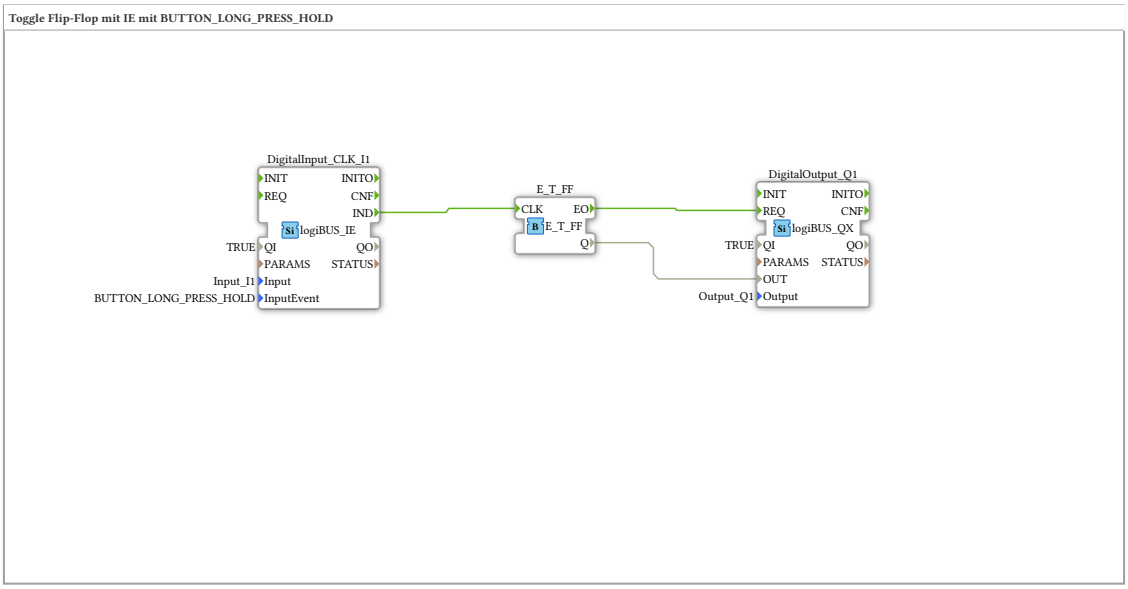
Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_004c4`.

Ziel der Übung

Nutzung des Ereignisses `BUTTON_LONG_PRESS_HOLD`.

Funktionsweise

Der Baustein `DigitalInput_CLK_I1` in `Uebung_004c4.SUB` ist auf permanentes Halten konfiguriert.



Dieses Ereignis wird **periodisch wiederholt** (z.B. alle 200ms), solange der Taster nach der ersten Erkennung des langen Drucks weiterhin gehalten wird. Da in dieser Übung ein Toggle-Flip-Flop am Ausgang hängt, führt dies dazu, dass die Lampe schnell an- und ausgeht (Blinken), solange der Finger auf dem Taster ist.

Anwendungsbeispiel

Wert-Inkrementierung: Solange man eine Taste gedrückt hält, zählt ein Wert (z.B. die Zieltemperatur oder die Motordrehzahl) kontinuierlich hoch.

2.2.1.70 Uebung_004c5: Toggle Flip-Flop mit IE mit BUTTON_PRESS_END



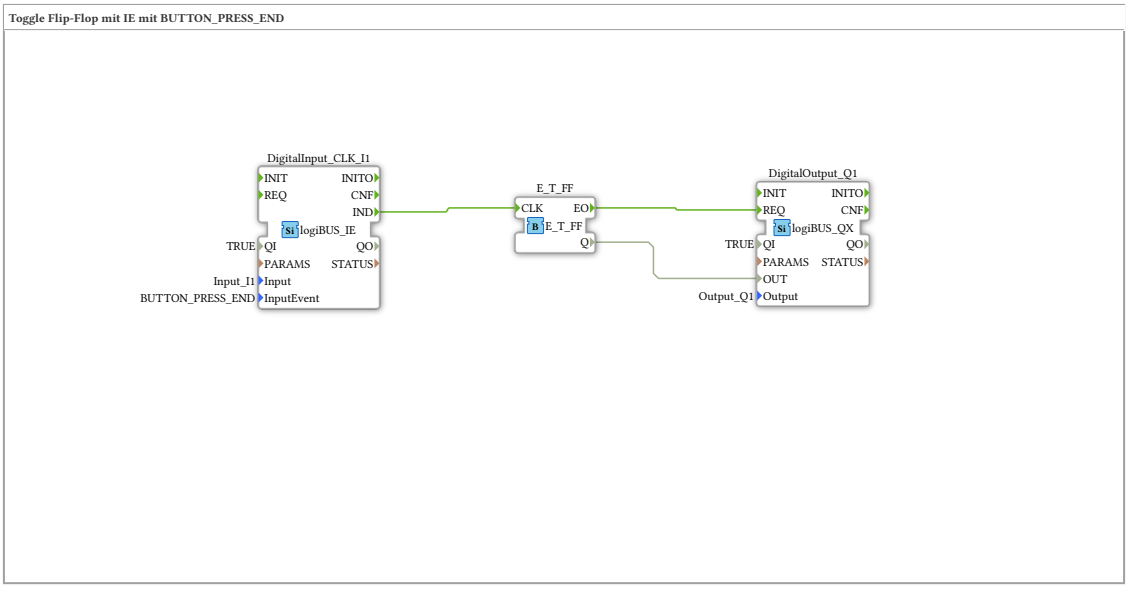
Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_004c5`.

Ziel der Übung

Nutzung des Ereignisses `BUTTON_PRESS_END`.

Funktionsweise

Der Baustein `DigitalInput_CLK_I1` in `Uebung_004c5.SUB` reagiert auf jede fallende Flanke.



Dieses Ereignis feuert **immer**, wenn der Taster losgelassen wird – völlig unabhängig davon, ob er vorher kurz (`CLICK`) oder lang (`LONG_PRESS`) gedrückt wurde. Es ist das universelle Ereignis für das Ende einer Interaktion.

Anwendungsbeispiel

Sicherheits-Stopp: Eine Funktion (z.B. ein Kranarm) bewegt sich, solange die Taste gedrückt ist. Sobald der Finger weggenommen wird (`PRESS_END`), muss die Bewegung sofort stoppen, egal wie kurz die Betätigung war.

2.2.1.71 Uebung_004c6: Toggle Flip-Flop mit IE mit BUTTON_MULTIPLE_CLICK (3-fach Klick)

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_004c6`. Hier wird der erweiterte Baustein `logiBUS_IE2` genutzt, um eine spezifische Anzahl von Klicks auszuwerten.

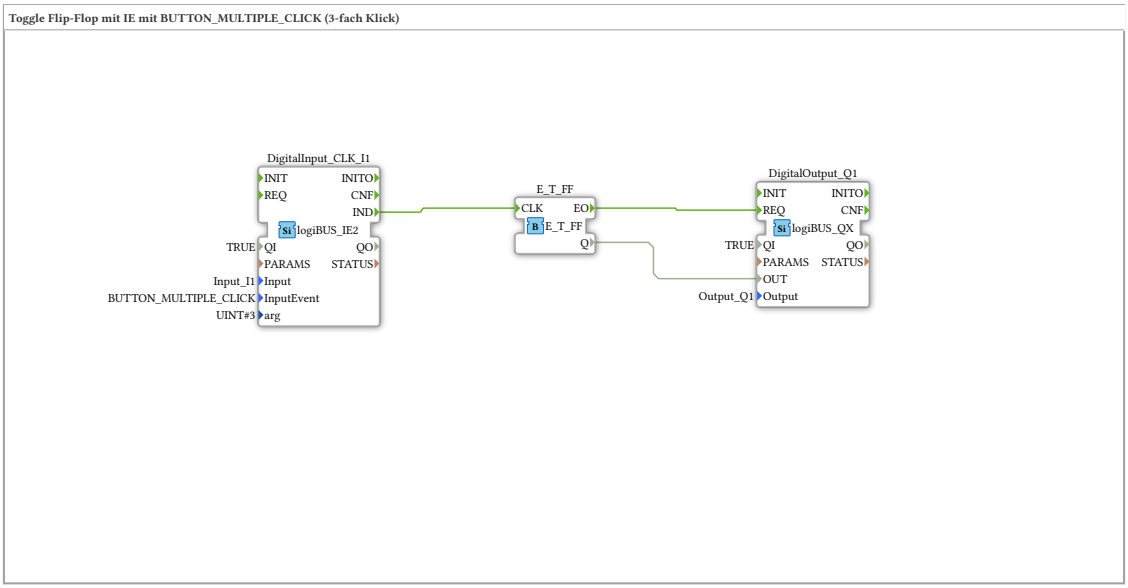
Ziel der Übung

Konfiguration eines n-fach Klicks unter Verwendung von Argumenten.

Beschreibung und Komponenten

Die Subapplikation `Uebung_004c6.SUB` nutzt den Bausteintyp `logiBUS_IE2` mit dem Ereignis `BUTTON_MULTIPLE_CLICK` und dem Argument `arg = 3`.

Funktionsweise



Der Baustein zählt die Klicks innerhalb eines Zeitfensters. Nur wenn der Nutzer den Taster **exakt dreimal** kurz hintereinander drückt, wird das Ereignis `IND` gefeuert und die Lampe toggelt. Alle anderen Klick-Kombinationen werden verworfen.

Anwendungsbeispiel

Versteckte Experten-Funktionen: Zugriff auf Kalibrierungs-Modi oder Service-Menüs, die für den normalen Anwender nicht direkt ersichtlich sein sollen. Ein Triple-Click ist eine bewusste Handlung, die im normalen Betrieb kaum versehentlich vorkommt.

2.2.1.72 Uebung_004c7: Toggle Flip-Flop mit IE mit BUTTON_PRESS_END

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_004c7`. Auch hier wird der Baustein `logiBUS_IE2` genutzt, um die Haltezeit für ein Ereignis individuell anzupassen.

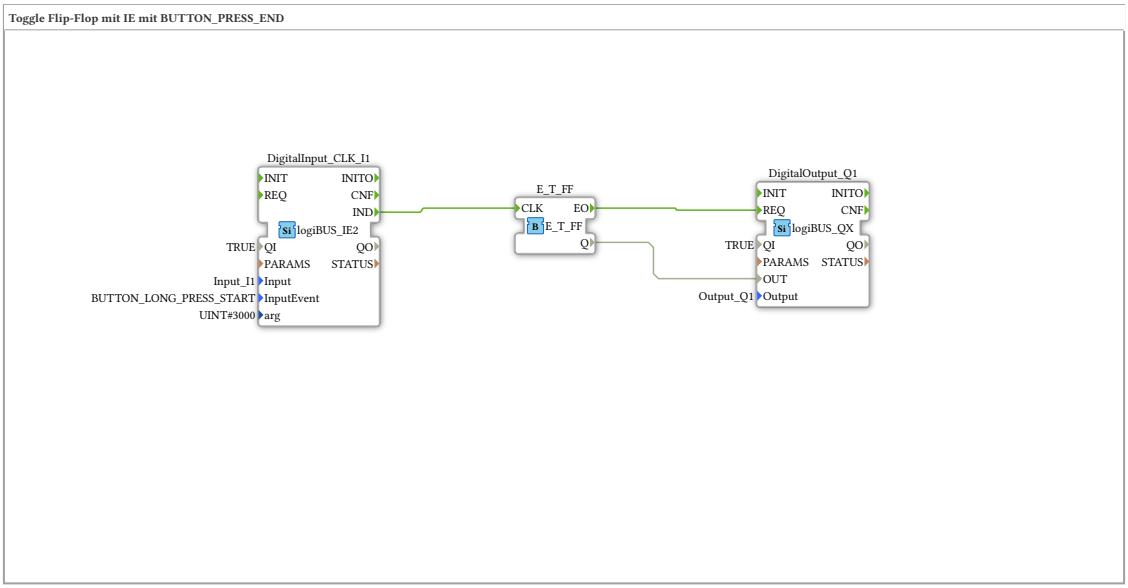
Ziel der Übung

Festlegung einer spezifischen Zeitdauer für einen langen Tastendruck.

Beschreibung und Komponenten

Die Subapplikation `Uebung_004c7.SUB` nutzt `logiBUS_IE2` mit `BUTTON_LONG_PRESS_START` und dem Argument `arg = 3000`.

Funktionsweise

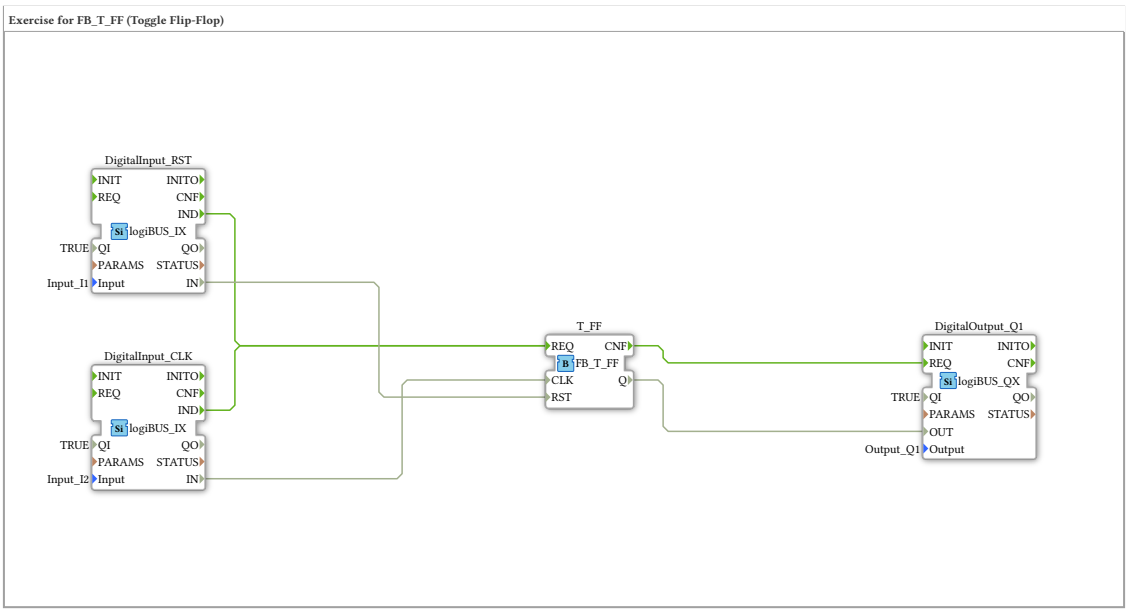


Die Einheit des Arguments `arg` sind Millisekunden. Das bedeutet: Das Ereignis `IND` wird erst dann gefeuert, wenn der Taster für **mindestens 3 Sekunden** (3000ms) ununterbrochen gedrückt wurde. Dies überschreibt den im System hinterlegten Standardwert für „Long Press“.

Anwendungsbeispiel

Werkseinstellungen laden (Factory Reset): Eine kritische Aktion, die eine sehr bewusste und lange Interaktion des Nutzers erfordert, um versehentliche Datenverluste absolut sicher auszuschließen.

2.2.1.73 Uebung_004d_T: Exercise for FB_T_FF (Toggle Flip-Flop)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert die Anwendung des Funktionsbausteins **FB_T_FF** (Toggle Flip-Flop). Ein Toggle-Flipflop ändert seinen Ausgangszustand bei jedem positiven Flanke des Taktsignals (CLK). Zusätzlich kann der Ausgang durch ein Resetsignal (RST) zurückgesetzt werden. In der Übung werden zwei digitale Eingänge als Takt- und Reset-Quelle verwendet, der T-FF-Baustein verarbeitet die Signale und

steuert einen digitalen Ausgang. Ziel ist es, das grundlegende Umschaltverhalten eines Toggle-Flipflops in einer Automatisierungsumgebung zu verstehen.

Verwendete Funktionsbausteine (FBs)

Die Übung besteht aus vier Funktionsbausteinen, die im SubApp-Netzwerk miteinander verbunden sind:

DigitalInput_RST (logiBUS_IX)

- **Typ**: `logiBUS::io::DI::logiBUS_IX`
- **Parameter**: `QI = TRUE` (Qualifier für Initialisierung aktiv) `Input = Input_I1` (physischer Eingang der logiBUS-Klemme)
- **Funktionsweise**: Liest den digitalen Eingang `Input_I1` aus. Das Ereignis `IND` wird bei Signaländerung ausgelöst. Der gelesene Wert wird am Datenausgang `IN` bereitgestellt. Dient als Reset-Signal für das T-FF.

DigitalInput_CLK (logiBUS_IX)

- **Typ**: `logiBUS::io::DI::logiBUS_IX`
- **Parameter**: `QI = TRUE` `Input = Input_I2`
- **Funktionsweise**: Liest den digitalen Eingang `Input_I2` aus. Das Ereignis `IND` wird bei Signaländerung ausgelöst. Der gelesene Wert wird am Datenausgang `IN` bereitgestellt. Dient als Taktsignal für das T-FF.

T_FF (FB_T_FF)

- **Typ**: `logiBUS::bistableElements::FB_T_FF`
- **Parameter**: Keine expliziten Parameter, die Schnittstellen werden über Verbindungen belegt.
- **Funktionsweise**: Implementiert ein Toggle-Flipflop.
  - **Ereigniseingang `REQ`**: Startet die Verarbeitung.
  - **Dateneingang `CLK`**: Taktsignal (boolesch). Bei positiver Flanke von `CLK` wird der interne Zustand getoggelt.
  - **Dateneingang `RST`**: Resetsignal (boolesch). Bei `TRUE` wird der Ausgang sofort auf `FALSE` gesetzt, unabhängig vom Takt.
  - **Datenausgang `Q`**: Aktueller Zustand des Flipflops.
  - **Ereignisausgang `CNF`**: Wird nach abgeschlossener Verarbeitung ausgelöst.

DigitalOutput_Q1 (logiBUS_QX)

- **Typ**: `logiBUS::io::DQ::logiBUS_QX`
- **Parameter**: `QI = TRUE` `Output = Output_Q1`
- **Funktionsweise**: Empfängt den Zustand des T-FF über den Dateneingang `OUT` und gibt diesen am physischen Ausgang `Output_Q1` aus. Der Ausgang wird durch das Ereignis `REQ` aktualisiert.

Programmablauf und Verbindungen

1. **Ereignisverkettung**:
   - Beide Digitaleingänge (`DigitalInput_RST` und `DigitalInput_CLK`) sind über ihren Ereignisausgang `IND` mit dem Ereigniseingang `REQ` des `T_FF` verbunden.
   - Das bedeutet: Jede Änderung eines der beiden Eingänge löst eine Verarbeitung des T-FF aus.
   - Nach der Verarbeitung des T-FF wird der Ereignisausgang `CNF` mit dem `REQ`-Eingang des `DigitalOutput_Q1` verbunden, sodass der Ausgangswert sofort an die Hardware weitergereicht wird.
2. **Datenverkettung**:
   - Der gelesene Wert des Reseteingangs (`DigitalInput_RST.IN`) wird auf den `RST`-Eingang des T-FF gelegt.
   - Der gelesene Wert des Takteingangs (`DigitalInput_CLK.IN`) wird auf den `CLK`-Eingang des T-FF gelegt.
   - Der Ausgang des T-FF (`T_FF.Q`) wird auf den `OUT`-Eingang des Digitalausgangs `DigitalOutput_Q1` verbunden.
3. **Funktionsweise im Betrieb**:
   - Wird `Input_I2` (Takt) von `FALSE` auf `TRUE` geändert (positive Flanke), toggelt der Ausgang `Q` des T-FF.

- Wird gleichzeitig oder später `Input_I1` (Reset) auf `TRUE` gesetzt, wird `Q` sofort `FALSE` (asynchroner Reset).
- Der aktuelle Zustand von `Q` erscheint am Ausgang `Output_Q1`.

Lernziele:

- Funktionsweise eines Toggle-Flipflops (T-FF) verstehen.
- Ereignisgesteuerte Datenflussmodellierung in 4diac (IEC 61499).
- Einfache Verbindung von Hardware-Eingängen/Ausgängen mit logischen Bausteinen.

Schwierigkeitsgrad: Einfach (grundlegende Übung). **Vorkenntnisse:** Grundlagen der Digitaltechnik, Einführung in die 4diac-IDE.

Start der Übung:

1. Laden Sie das Projekt in die 4diac-IDE (die SubApp `Uebung_004d_T` ist in der Klasse `Uebungen` enthalten).
2. Weisen Sie die Eingänge `Input_I1` und `Input_I2` sowie den Ausgang `Output_Q1` den entsprechenden logiBUS-Klemmen Ihrer Hardware zu.
3. Starten Sie die Anwendung und beobachten Sie das Verhalten durch Anlegen von Signalen an die Eingänge.

Zusammenfassung

Die Übung `Uebung_004d_T` zeigt den Einsatz des Funktionsbausteins `FB_T_FF` zur Realisierung eines Toggle-Flipflops. Durch die Kopplung von digitalen Eingängen, dem T-FF und einem digitalen Ausgang wird das grundlegende Schaltverhalten – Toggeln bei Taktflanke und asynchrones Rücksetzen – vermittelt. Die einfache Ereignis- und Datenverkettung macht diese Übung zu einem idealen Einstieg in die Arbeit mit bistabilen Elementen unter IEC 61499.

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

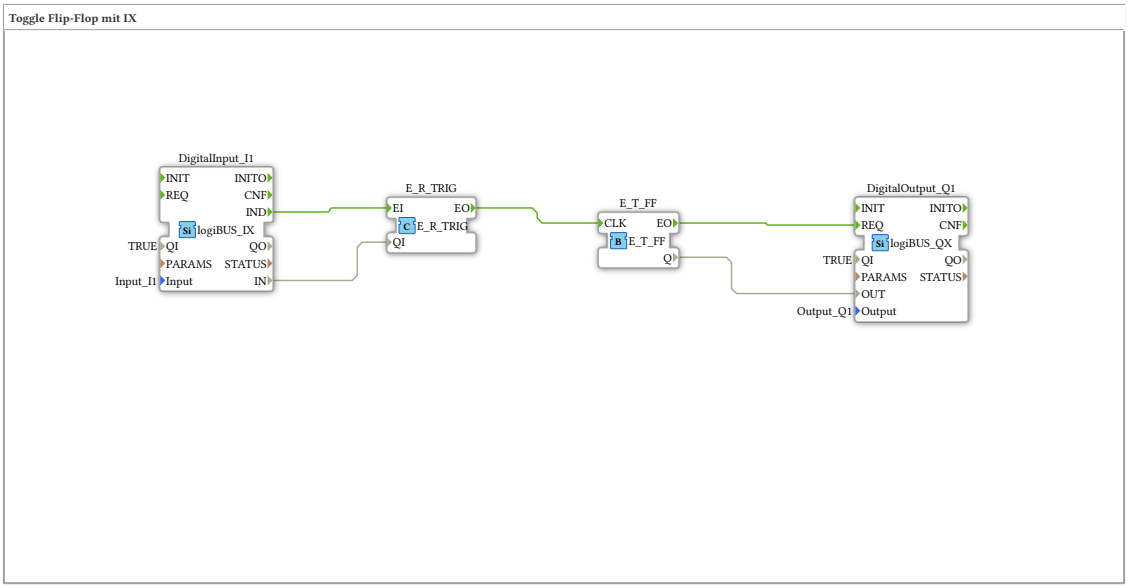
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.2.1.74 Uebung_005: Toggle Flip-Flop mit IX

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_005`. Hier wird demonstriert, wie ein zustandsbasierter Hardware-Eingang (`IX`) genutzt werden kann, um ein ereignisbasiertes Toggle-Flip-Flop zu steuern.

🎧 Podcast



- Automatisierung entschlüsselt: Leiten, Steuern, Regeln – Die unsichtbare Sprache der Technik (DIN IEC 60050-351)

Ziel der Übung

Verständnis der Flankenauswertung unter Verwendung von Ereignisweichen. Es wird gezeigt, wie man aus einem kontinuierlichen Signal (Taster gedrückt) einen einzelnen Impuls zum Umschalten generiert, ohne den spezialisierten `logiBUS_IE` Baustein zu verwenden.

Beschreibung und Komponenten

Die Subapplikation `Uebung_005.SUB` kombiniert einen Standard-Eingang (`IX`) mit einer Ereignis-Weiche, um ein Flip-Flop zu takten.

Funktionsbausteine (FBs)

- **`DigitalInput_I1`**: Typ `logiBUS_IX`. Liefert ein Ereignis bei jeder Pegeländerung (Drücken und Loslassen).
- **`E_SWITCH`**: Dient als Gatter, um nur eine der beiden Flanken durchzulassen.
- **`E_T_FF`**: Das Toggle-Flip-Flop.

Funktionsweise

Die Schaltung nutzt die Datenverbindung vom Eingang zum Gate der Weiche:

```
<EventConnections>
    <Connection Source="DigitalInput_I1.IND" Destination="E_SWITCH.EI"/>
    <Connection Source="E_SWITCH.EO1" Destination="E_T_FF.CLK"/>
</EventConnections>
<DataConnections>
    <Connection Source="DigitalInput_I1.IN" Destination="E_SWITCH.G"/>
</DataConnections>
```

Der funktionale Ablauf:

1. **Drücken**: `I1` wechselt von FALSE auf TRUE. Ein `IND`-Event wird gesendet. Da am Eingang `G` der Weiche nun TRUE anliegt, wird das Event an `EO1` ➡️ `CLK` weitergeleitet. Das Licht toggelt.
2. **Loslassen**: `I1` wechselt zurück auf FALSE. Wieder wird ein `IND`-Event gesendet. Da am Eingang `G` nun aber FALSE anliegt, wird das Event an `EO0` (hier nicht verbunden) geleitet. Das Flip-Flop reagiert nicht.

Ergebnis: Die Lampe schaltet nur beim Drücken des Tasters um (steigende Flanke).

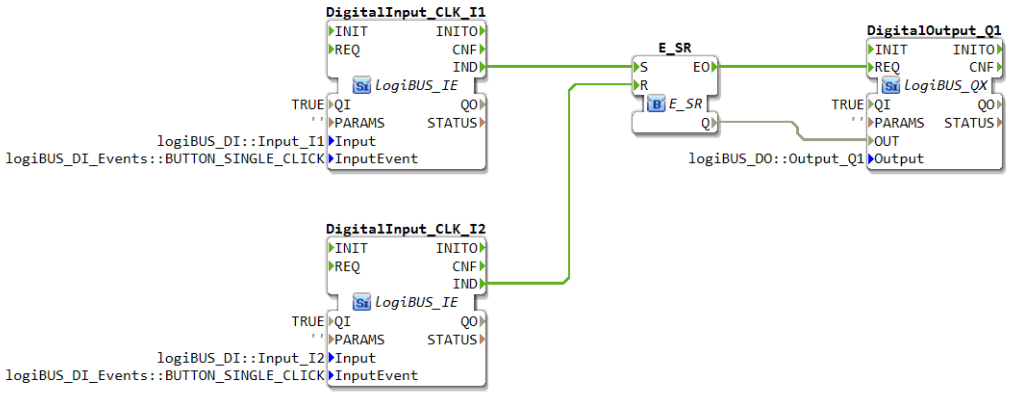
Bewertung

Dieser Aufbau verdeutlicht die Interaktion von Daten (`G`) und Events (`EI`). In der Praxis ist für diese Aufgabe jedoch die Verwendung eines `logiBUS_IE` Bausteins (siehe Übung 004a) effizienter.

2.2.1.75 Uebung_006: SR-Flip-Flop mit 2x IE



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_006`. Hier wird ein klassischer Selbsthaltespeicher mit getrennten Tastern für Ein und Aus implementiert.



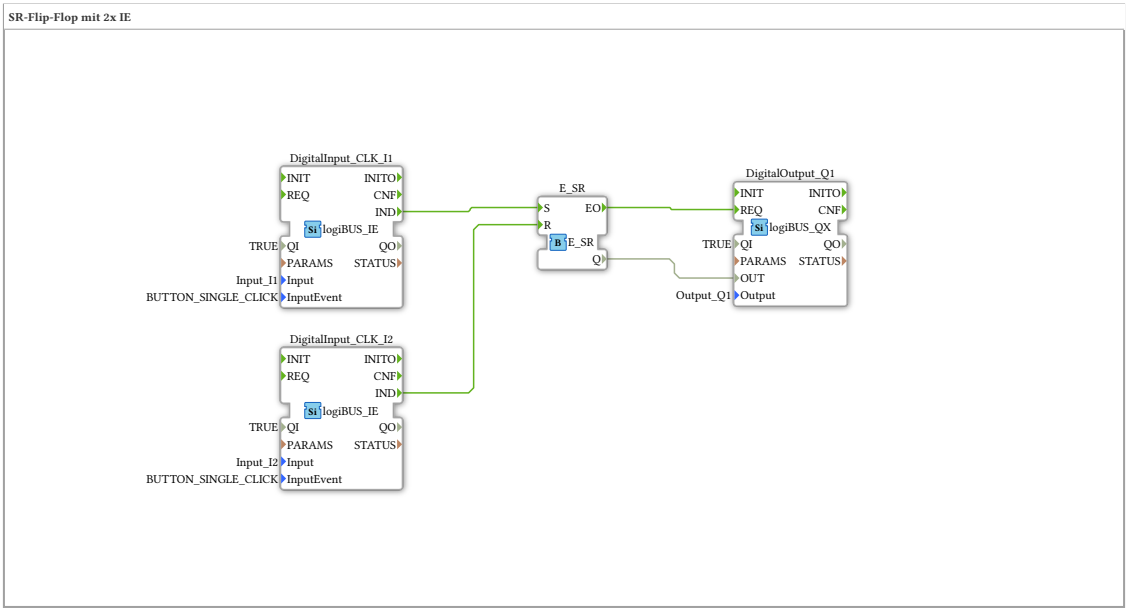
Ziel der Übung

Realisierung einer Schaltung mit getrennter Setz- und Rücksetz-Logik unter Verwendung von Ereignis-Bausteinen.

Beschreibung und Komponenten

Die Subapplikation `Uebung_006.SUB` nutzt zwei ereignisbasierte Eingänge und einen SR-Speicher.

Funktionsbausteine (FBs)



- **`I1` (Set)**: Taster zum Einschalten (konfiguriert auf Einzelklick).
- **`I2` (Reset)**: Taster zum Ausschalten (konfiguriert auf Einzelklick).
- **`E_SR`**: Ein ereignisbasierter Speicherbaustein. Ein Event am Eingang `S` (Set) setzt den Ausgang `Q` auf TRUE, ein Event am Eingang `R` (Reset) setzt ihn auf FALSE.

Funktionsweise

```
<EventConnections>
    <Connection Source="DigitalInput_CLK_I1.IND" Destination="E_SR.S"/>
    <Connection Source="DigitalInput_CLK_I2.IND" Destination="E_SR.R"/>
    <Connection Source="E_SR.EO" Destination="DigitalOutput_Q1.REQ"/>
</EventConnections>
```

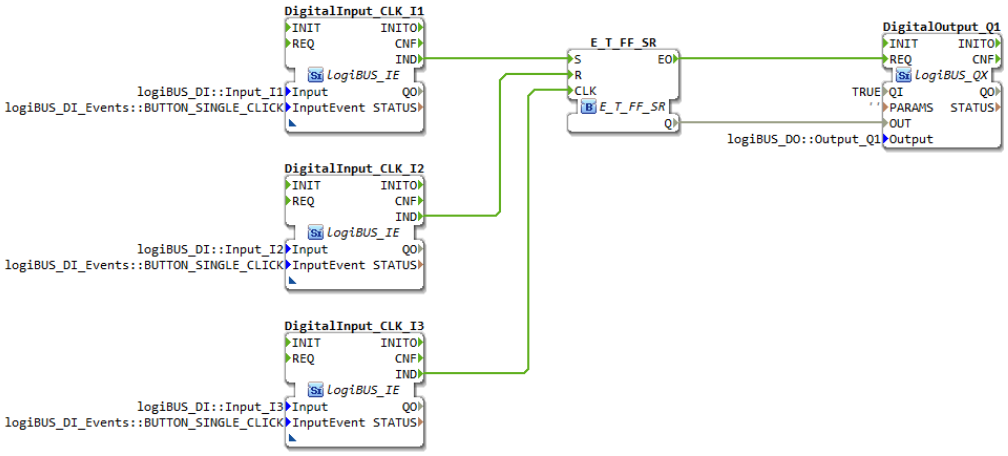
- Ein Klick auf Taster 1 ➡️ Speicher wird gesetzt ➡️ Lampe geht an.
- Ein Klick auf Taster 2 ➡️ Speicher wird gelöscht ➡️ Lampe geht aus.
- Erneutes Drücken von Taster 1, wenn das Licht bereits an ist, hat keine Auswirkung.

Anwendungsbeispiel

Industrielle Start/Stopp-Steuerung: Ein grüner Taster startet eine Maschine, ein roter Taster stoppt sie. Dies ist sicherer als ein einzelner Toggle-Taster, da der Bediener immer einen definierten Befehl („Ich will aus“) geben kann, unabhängig vom aktuellen Zustand.

2.2.1.76 Uebung_006a: SR und T-Flip-Flop mit 3x IE

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_006a`. Hier wird ein hochflexibler Speicherbaustein verwendet, der drei verschiedene Bedienweisen kombiniert.



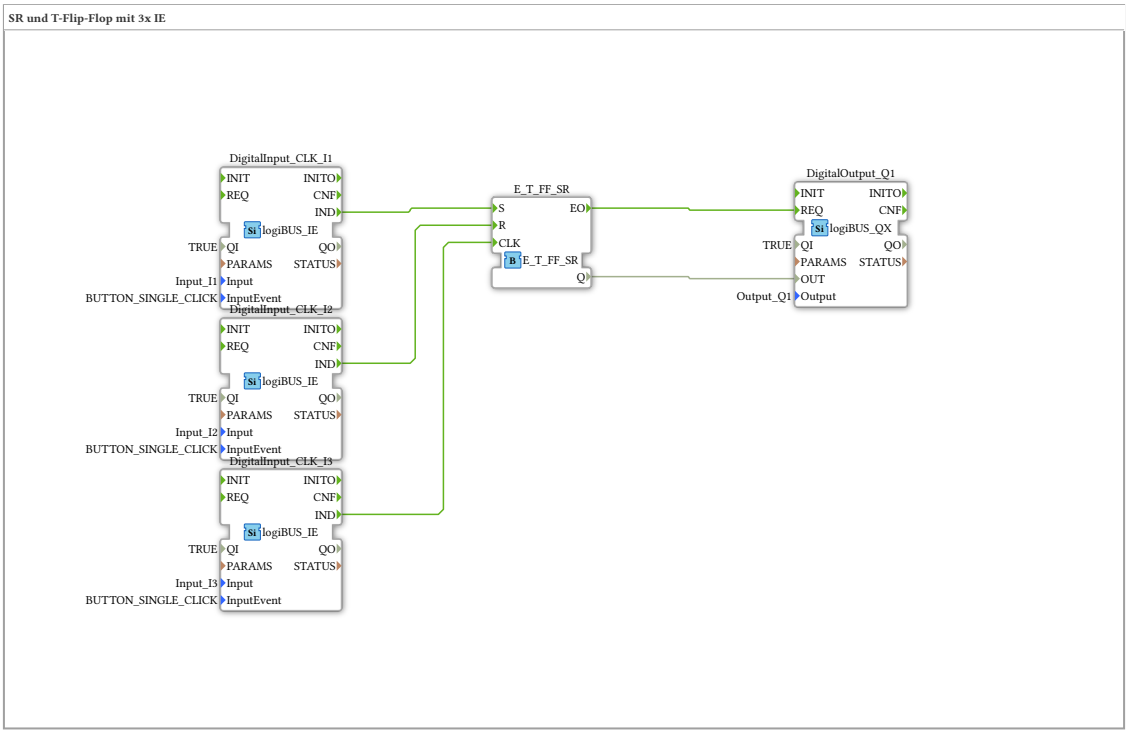
Ziel der Übung

Einführung des `E_T_FF_SR` Bausteins. Dieser vereint die Funktionen eines Toggle-Flip-Flops mit denen eines SR-Speichers.

Beschreibung und Komponenten

Die Subapplikation `Uebung_006a.SUB` verknüpft drei separate Taster mit einem zentralen Speicherglied.

Funktionsbausteine (FBs)



- **`I1` (Set)**: Schaltet den Ausgang ein.
- **`I2` (Reset)**: Schaltet den Ausgang aus.
- **`I3` (Toggle)**: Wechselt den aktuellen Zustand.
- **`E_T_FF_SR`**: Der Kombi-Baustein für alle drei Ereignistypen.

Funktionsweise

Der Baustein reagiert auf jedes der drei Eingangs-Events individuell:

- Ein Event an `S` setzt den Zustand fest auf `TRUE`.
- Ein Event an `R` setzt den Zustand fest auf `FALSE`.
- Ein Event an `CLK` invertiert den aktuellen Zustand (Toggle).

Alle Ereignisse führen zu einer Aktualisierung des Ausgangs `Q` und feuern das Bestätigungs-Event `EO` ab, um die Hardware anzusteuern.

Anwendungsbeispiel

Gebäude-Lichtsteuerung:

- **Vor Ort**: Ein Taster im Zimmer toggelt das Licht (`I3`).
- **Zentrale**: Am Hauseingang gibt es einen Taster „Gute Nacht“, der alle Lichter per Reset (`I2`) ausschaltet.
- **Alarmanlage**: Im Falle eines Einbruchs setzt die Zentrale alle Lichter per Set (`I1`) dauerhaft ein.

2.2.1.77 Uebung_006a2: 2x SR und T-Flip-Flop mit IE

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_006a2`.

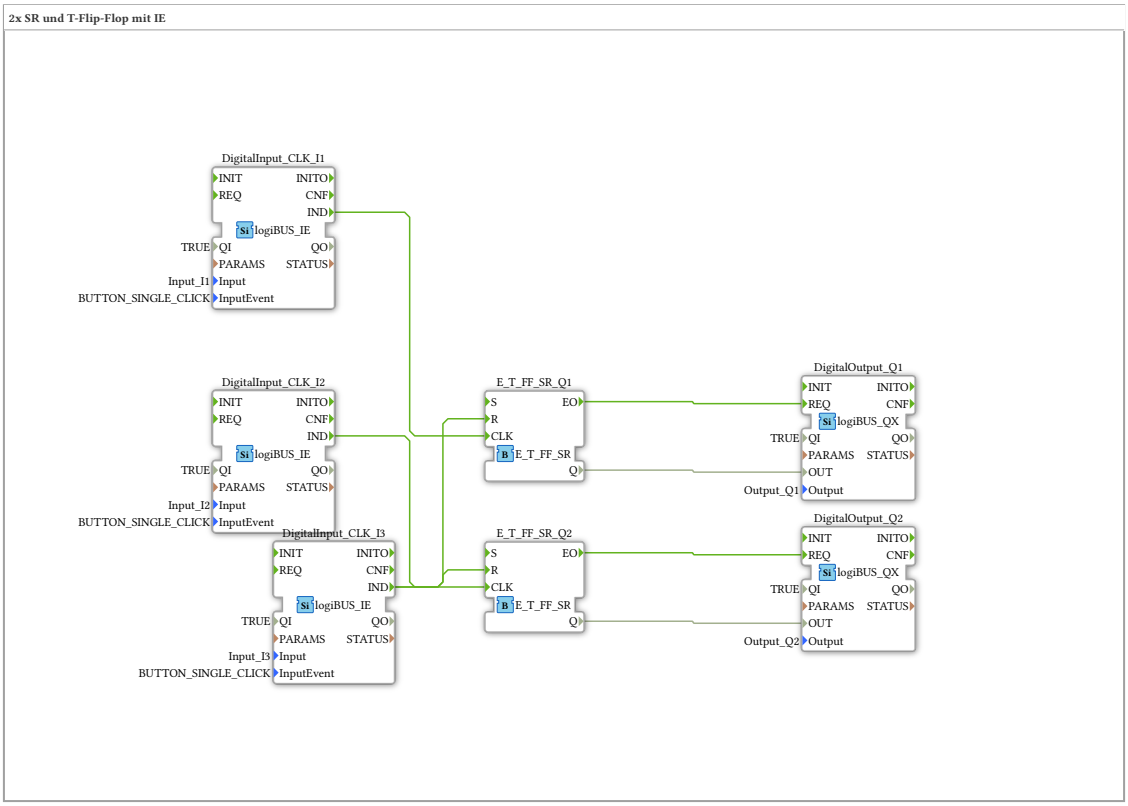
Ziel der Übung

Realisierung einer „Zentral-Aus“ Funktion für mehrere unabhängige Speicherglieder.

Beschreibung und Komponenten

Die Subapplikation `Uebung_006a2.SUB` steuert zwei separate Lampen (`Q1`, `Q2`) über zwei Taster (`I1`, `I2`), die durch einen dritten Taster (`I3`) gemeinsam zurückgesetzt werden können.

Funktionsbausteine (FBs)



- **2x `E_T_FF_SR`**: Einer für jeden Lichtkanal.
- **`I1` & `I2`**: Tasten zum individuellen Umschalten der Kanäle.
- **`I3`**: Gemeinsamer Reset-Taster.

Funktionsweise

Die Logik nutzt das Fan-Out Prinzip für Ereignisse:

- `I1` ist mit `CLK` von Flip-Flop 1 verbunden.
- `I2` ist mit `CLK` von Flip-Flop 2 verbunden.
- `I3` (Reset) ist mit den `R`-Eingängen **beider** Flip-Flops verbunden.

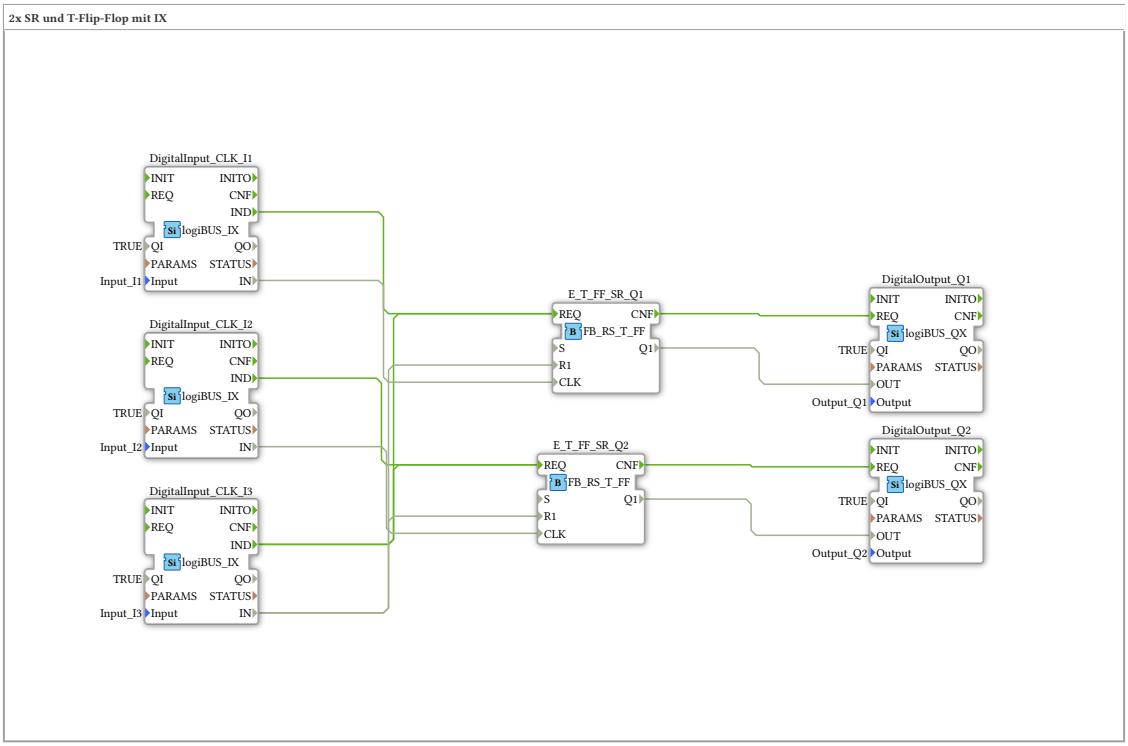
Ein Druck auf `I3` schaltet sofort alle Lampen im System aus, unabhängig davon, in welchem Zustand sie sich vorher befanden.

Anwendungsbeispiel

Werkstatt-Beleuchtung: Jede Maschine hat ihr eigenes Arbeitslicht. Am Ende des Arbeitstages kann der Werkstattleiter über einen zentralen Schalter an der Tür alle Lichter gleichzeitig löschen.

2.2.1.78 Uebung_006a2b: 2x SR und T-Flip-Flop mit IX

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_006a2b (2x SR und T-Flip-Flop mit IX).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **2x SR und T-Flip-Flop mit IX**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_006a2b.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QX.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
- **DigitalInput_CLK_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IX.
  - Parameter QI = TRUE
  - Parameter PARAMS =
  - Parameter Input = Input_I1
- **DigitalInput_CLK_I2**: Instanz des Typs logiBUS::io::DI::logiBUS_IX.
  - Parameter QI = TRUE
  - Parameter PARAMS =
  - Parameter Input = Input_I2
- **E_T_FF_SR_Q1**: Instanz des Typs logiBUS::bistableElements::FB_RS_T_FF.
- **DigitalInput_CLK_I3**: Instanz des Typs logiBUS::io::DI::logiBUS_IX.
  - Parameter QI = TRUE
  - Parameter PARAMS =
  - Parameter Input = Input_I3
- **E_T_FF_SR_Q2**: Instanz des Typs logiBUS::bistableElements::FB_RS_T_FF.
- **DigitalOutput_Q2**: Instanz des Typs logiBUS::io::DQ::logiBUS_QX.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q2

Verbindungen und Schnittstellen

Ereignisverbindungen:

- DigitalInput_CLK_I1.IND -> E_T_FF_SR_Q1.REQ
- DigitalInput_CLK_I2.IND -> E_T_FF_SR_Q2.REQ

- DigitalInput_CLK_I3.IND -> E_T_FF_SR_Q1.REQ
- DigitalInput_CLK_I3.IND -> E_T_FF_SR_Q2.REQ
- E_T_FF_SR_Q1.CNF -> DigitalOutput_Q1.REQ
- E_T_FF_SR_Q2.CNF -> DigitalOutput_Q2.REQ

Datenverbindungen:

- DigitalInput_CLK_I1.IN -> E_T_FF_SR_Q1.CLK
- DigitalInput_CLK_I2.IN -> E_T_FF_SR_Q2.CLK
- DigitalInput_CLK_I3.IN -> E_T_FF_SR_Q1.R1
- DigitalInput_CLK_I3.IN -> E_T_FF_SR_Q2.R1
- E_T_FF_SR_Q1.Q1 -> DigitalOutput_Q1.OUT
- E_T_FF_SR_Q2.Q1 -> DigitalOutput_Q2.OUT

Hinweise aus dem Modell

Hausmeister-Aus (alles Aus mit einem Druck)

Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_006a2b demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.2.1.79 Uebung_006a3: SR und T-Flip-Flop mit 3x IE



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_006a3`. Dies ist eine anspruchsvollere Anwendung zur Steuerung eines Motors mit zwei Drehrichtungen und automatischer Umschaltung.

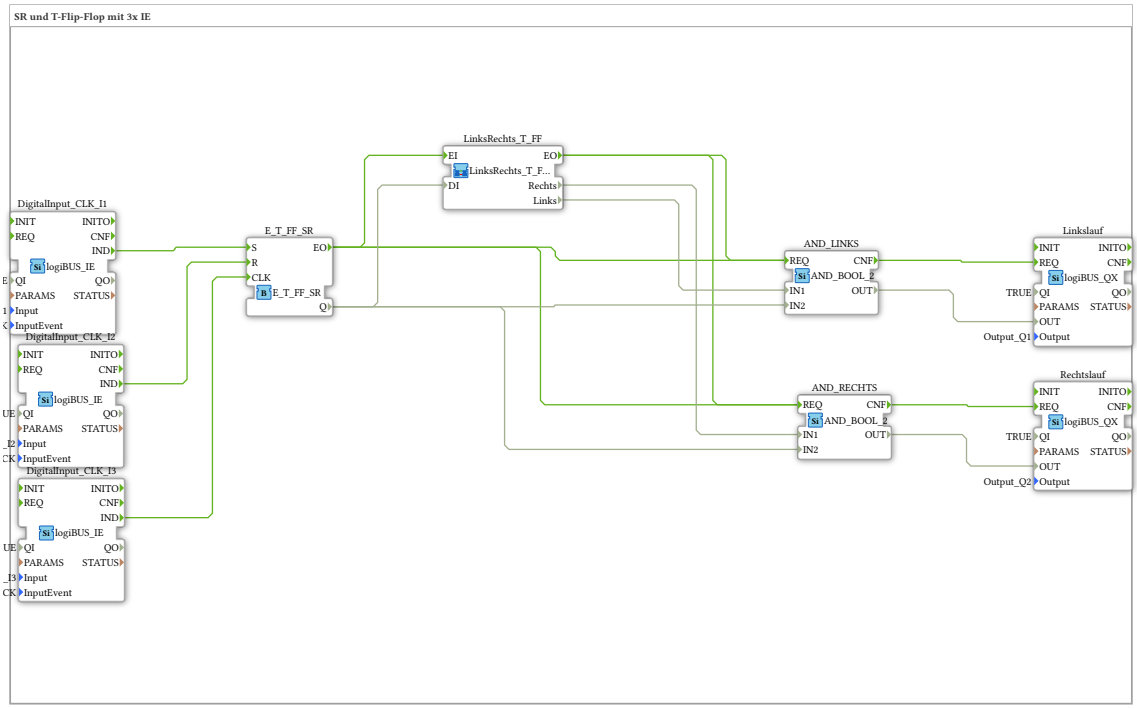
Ziel der Übung

Aufbau einer Steuerung für Vorwärts- und Rückwärtslauf mit Software-Verriegelung. Es muss sichergestellt werden, dass niemals beide Richtungen gleichzeitig angesteuert werden können.

Beschreibung und Komponenten

Die Subapplikation `Uebung_006a3.SUB` kombiniert einen Haupt-Ein/Aus-Speicher mit einer Logik zur Richtungsentscheidung.

Funktionsbausteine (FBs)



- **`E_T_FF_SR`**: Bestimmt, ob der Motor läuft (Ein/Aus).
- **`LinksRechts_T_FF` (SubApp)**: Ein interner Merker, der bei jedem Start die Richtung wechselt.
- **2x `AND_2_BOOL`**: Verknüpfen das „Ein“-Signal mit der gewählten Richtung.
- **`Q1` (Linkslauf) & `Q2` (Rechtslauf)**: Die Hardware-Ausgänge.

Funktionsweise

1. Der Nutzer startet das System über `I1`, `I2` oder `I3`.
2. Das Flip-Flop liefert ein „Global Ein“ Signal.
3. Die SubApp `LinksRechts_T_FF` entscheidet, welcher Zweig aktiv ist.
4. Durch die UND-Gatter kann das „Ein“-Signal nur zu einem der beiden Ausgänge durchdringen.

Diese Schaltung demonstriert, wie man komplexe Entscheidungen durch die Kombination von Basisfunktionen (Speicher, Logikgatter, Sub-Applikationen) löst.

Anwendungsbeispiel

Reversierendes Rührwerk: Ein Motor in einem Mischtank soll bei jedem Einschalten die Drehrichtung ändern, um eine bessere Durchmischung des Mediums zu erreichen. Die Software stellt dabei sicher, dass der Motor immer nur in eine Richtung Strom erhält.

2.2.1.80 Uebung_006a3_sub

Hinweis

Diese Übung existiert nicht mehr als eigene Übung. Der Baustein wurde in die Bibliothek verschoben und heißt jetzt LinksRechts_T_FF_Event. Beschreibung, Schnittstelle und Netzwerk stehen dort.

2.2.1.81 Uebung_006a4: SR und T-Flip-Flop mit 3x IE und LinksRechts Baustein



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_006a4`. Hier wird die Motorsteuerung aus der vorherigen Übung durch die Verwendung eines fertigen Bibliotheksbausteins optimiert.

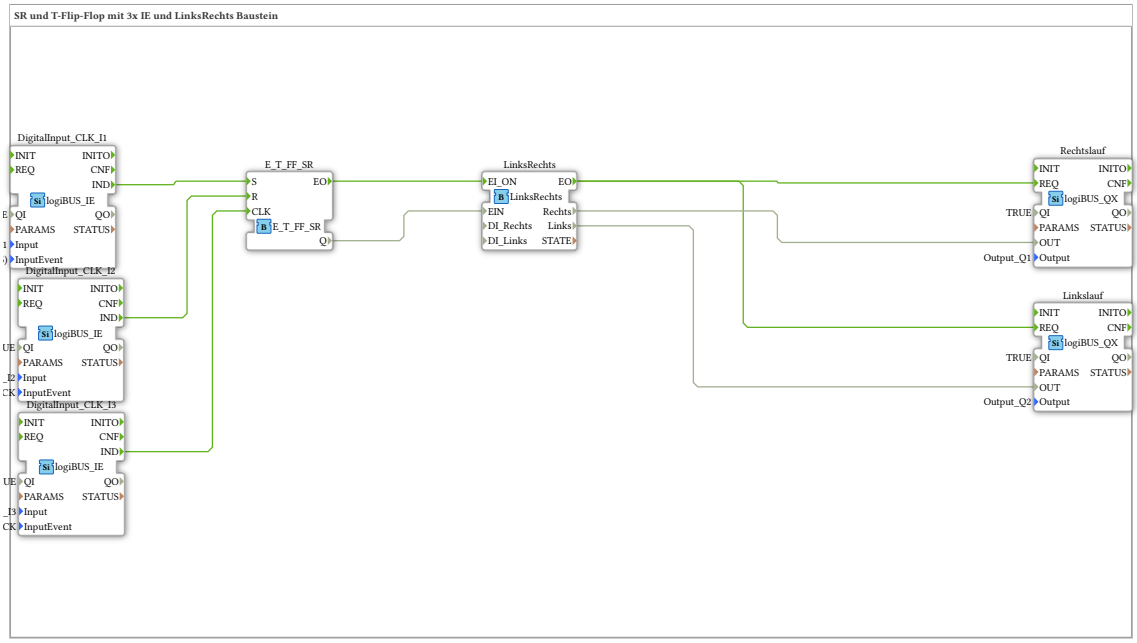
Ziel der Übung

Nutzung von spezialisierten Dienstbausteinen zur Reduktion der Diagramm-Komplexität.

Beschreibung und Komponenten

In `Uebung_006a4.SUB` wird das Netzwerk aus Gattern und Sub-Apps durch den Baustein `LinksRechts` ersetzt.

Funktionsbausteine (FBs)



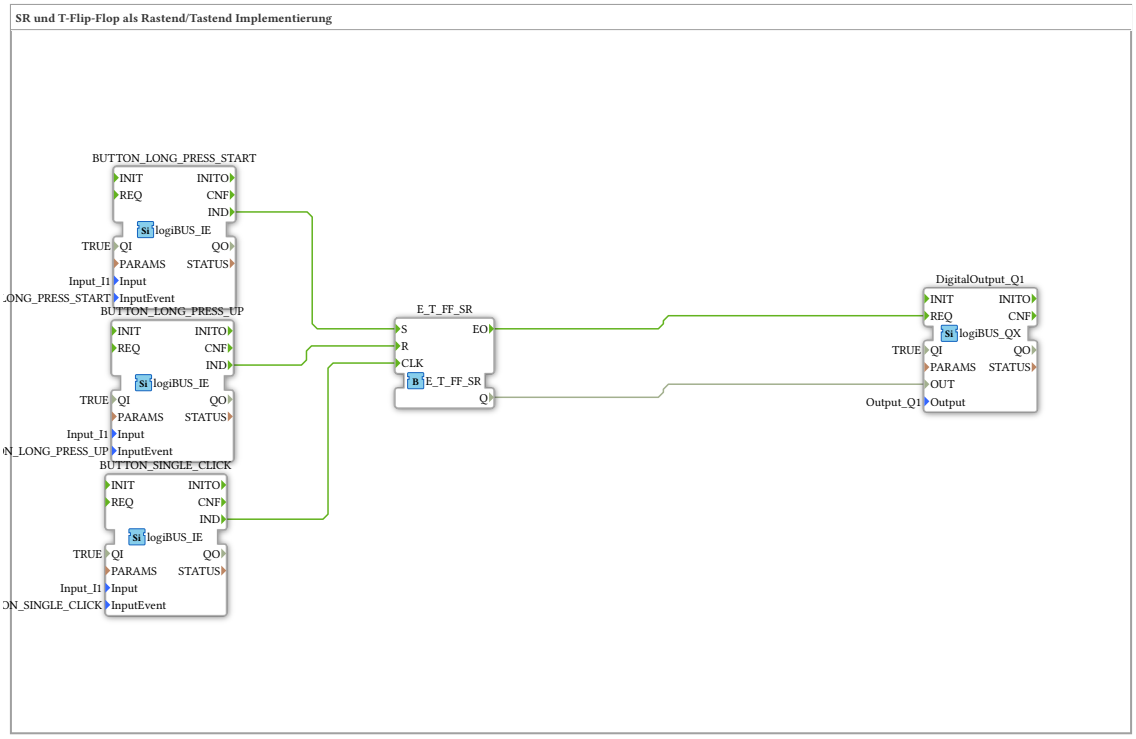
- **`LinksRechts`**: Typ `logiBUS::utils::sequence::verteiler::LinksRechts`. Dieser Baustein übernimmt die komplette Verwaltung der zwei Ausgänge inklusive der internen Richtungs-Logik.
- **`E_T_FF_SR`**: Liefert weiterhin das Startsignal an den Eingang `EI_ON`.

Vorteil

Durch die Verwendung von Bibliotheksbausteinen wird das Programm lesbarer und wartungsfreundlicher. Die interne Verriegelung ist im Baustein fest programmiert und kann nicht versehentlich durch fehlerhafte Verbindungen im Hauptdiagramm umgangen werden.

2.2.1.82 Uebung_006a5: SR und T-Flip-Flop als Rastend/Tastend Implementierung

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_006a5 (SR und T-Flip-Flop als Rastend/Tastend Implementierung).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **SR und T-Flip-Flop als Rastend/ Tastend Implementierung**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_006a5.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **BUTTON_LONG_PRESS_START**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter PARAMS =
  - Parameter Input = Input_I1
  - Parameter InputEvent = BUTTON_LONG_PRESS_START
- **BUTTON_LONG_PRESS_UP**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter PARAMS =
  - Parameter Input = Input_I1
  - Parameter InputEvent = BUTTON_LONG_PRESS_UP
- **E_T_FF_SR**: Instanz des Typs iec61499::events::E_T_FF_SR.
- **BUTTON_SINGLE_CLICK**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter PARAMS =
  - Parameter Input = Input_I1
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QX.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1

Verbindungen und Schnittstellen

Ereignisverbindungen:

- BUTTON_LONG_PRESS_START.IND -> E_T_FF_SR.S
- BUTTON_LONG_PRESS_UP.IND -> E_T_FF_SR.R

- BUTTON_SINGLE_CLICK.IND -> E_T_FF_SR.CLK
- E_T_FF_SR.EO -> DigitalOutput_Q1.REQ

Datenverbindungen:

- E_T_FF_SR.Q -> DigitalOutput_Q1.OUT

Hinweise aus dem Modell

Universal Eingang: so können wir mit einem Taster ODER einem Schalter arbeiten.

Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_006a5 demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.2.1.83 Uebung_006b: RS-Flip-Flop mit 2x IE



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_006b`.

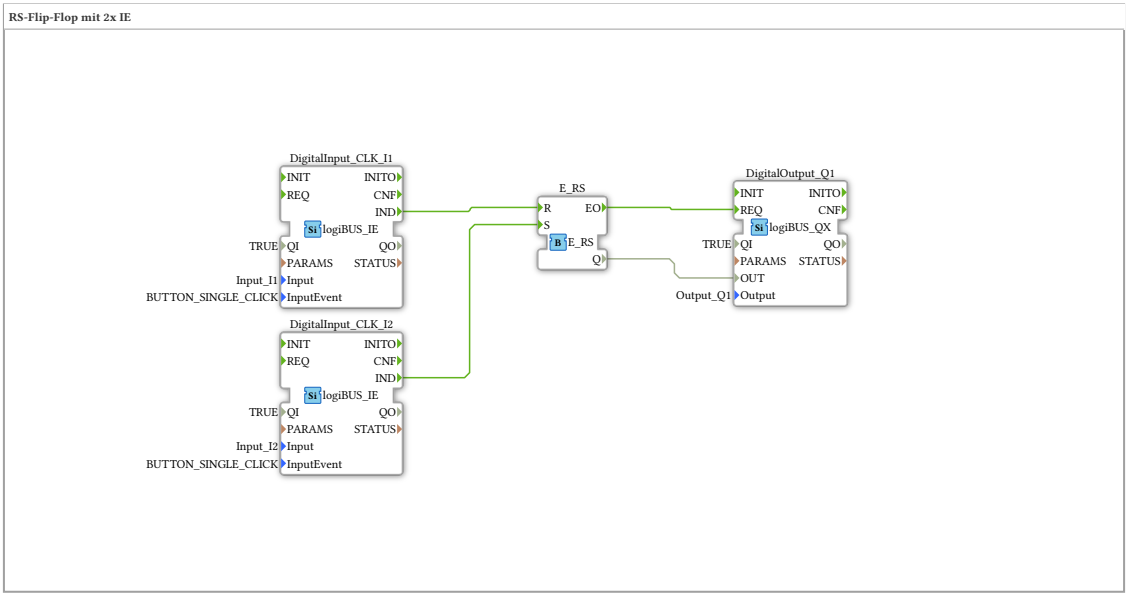
Ziel der Übung

Verständnis der Reset-Priorität.

Beschreibung und Komponenten

Die Subapplikation `Uebung_006b.SUB` nutzt einen `E_RS` Baustein.

Funktionsbausteine (FBs)



- **`E_RS`**: Ein ereignisbasiertes RS-Flip-Flop (Reset dominant).

Funktionsweise

Funktional sehr ähnlich zum SR-Speicher (Übung 006). Der entscheidende Unterschied liegt im Verhalten bei „Gleichzeitigkeit“: Sollten im exakt gleichen Moment sowohl ein Setz- als auch ein Rücksetz-Ereignis eintreffen, gewinnt beim `E_RS` das **Rücksetzen**. Der Ausgang wird also sicher auf `FALSE` geschaltet.

Anwendungsbeispiel

Gefahrenabschaltung: Bei einer Maschine hat der Stopp-Befehl immer Vorrang. Wenn ein Fehlerzustand den Stopp auslöst, darf ein gleichzeitiger Startversuch des Bedieners die Abschaltung nicht verhindern. Hier ist ein RS-Glied zwingend erforderlich.

2.2.1.84 Uebung_006c: SR-Flip-Flop mit IB auf DI_REPEAT



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_006c`. Hier wird eine komplexe Kanalsteuerung unter Verwendung von Byte-Daten und Ereignis-Demultiplexern realisiert.

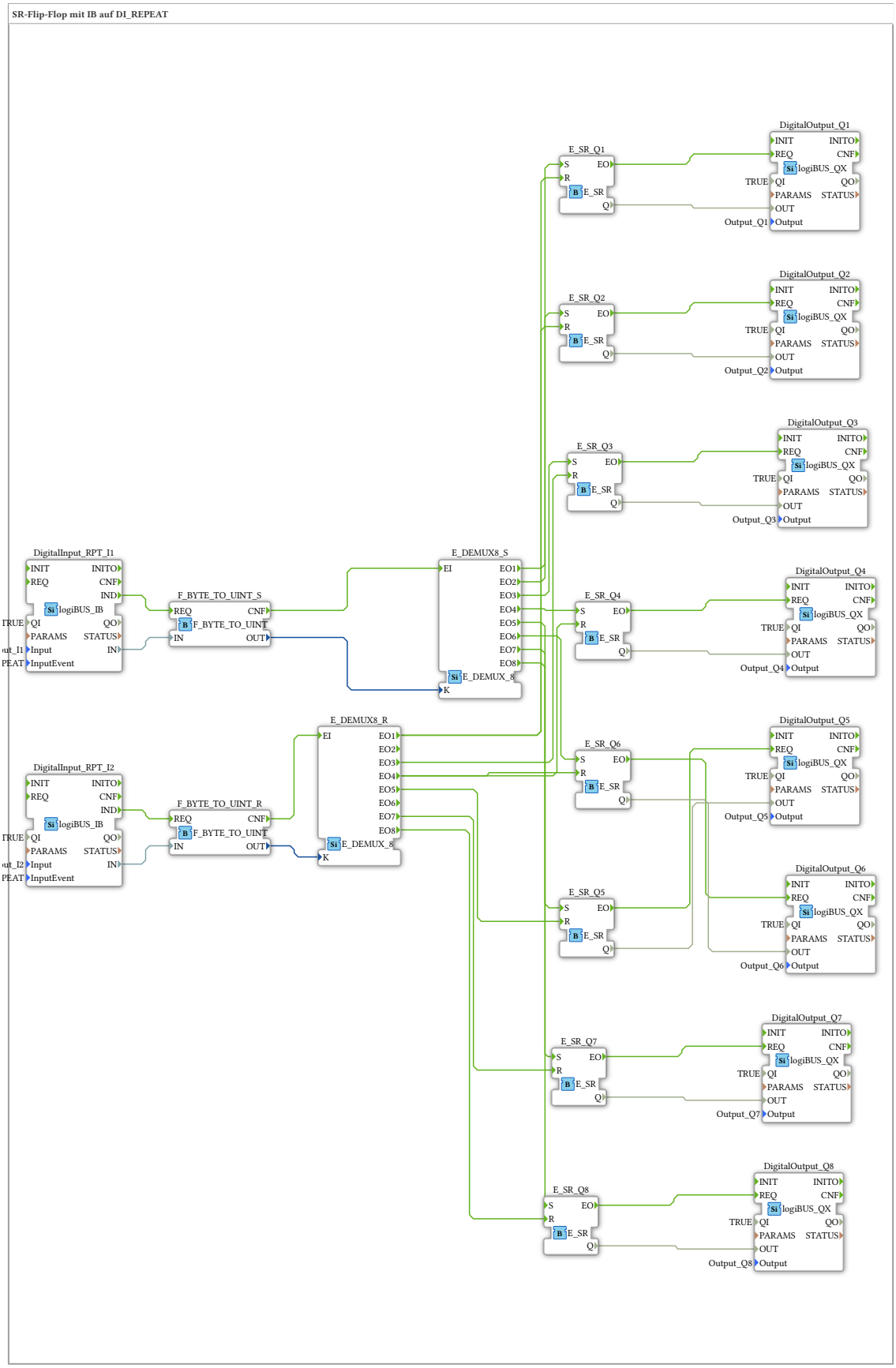
Ziel der Übung

Erlernen der adressierten Ereignisverteilung. Anstatt für jeden Kanal eine eigene Leitung zu ziehen, wird eine „Adresse“ (Index) genutzt, um ein Ereignis an das richtige Ziel zu leiten.

Beschreibung und Komponenten

Die Subapplikation `Uebung_006c.SUB` steuert 8 Lampenspeicher über zwei zentrale Wahlschalter.

Funktionsbausteine (FBs)



- **`logiBUS_IB`**: Ein spezieller Eingangsbaustein für „Input Byte“. Er liefert einen Zahlenwert (0-255), der meist von einem Multi-Funktions-Bedienelement (z.B. einem ISOBUS-Joystick mit vielen Tasten) stammt.
- **`E_DEMUX_8`**: Ein Ereignis-Demultiplexer. Er hat einen Ereignis-Eingang `EI` und einen Daten-Eingang `K` (Selector). Je nach Wert von `K` leitet er das Event an einen der acht Ausgänge `EO1` bis `EO8` weiter.
- **`8x E_SR`**: Speicher für die Ausgänge `Q1` bis `Q8`.

Funktionsweise

Das System arbeitet mit zwei Kanälen:

1. **Setzen-Kanal**: Ein Druck auf Taster `I1` (konfiguriert als Repeat) liefert eine Nummer. Der Demux `E_DEMUX8_S` leitet das Ereignis an den entsprechenden Speicherplatz weiter ➡️ Die Lampe geht an.
2. **Rücksetzen-Kanal**: Taster `I2` liefert analog dazu eine Nummer an `E_DEMUX8_R` ➡️ Die entsprechende Lampe geht aus.

Anwendungsbeispiel

Fernsteuerung von Aktoren über ein Terminal: Ein Bediener hat ein Nummernfeld oder ein Drehrad am Joystick. Er wählt eine Nummer aus und drückt „Aktivieren“. Die Steuerung sorgt dafür, dass genau das Gerät mit dieser Nummer eingeschaltet wird. Dies spart enorm viele physische Taster und Leitungen ein.

2.2.1.85 Uebung_006d: SR-Flip-Flop mit 2x IE und Doppelklick

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_006d`. Hier wird eine asymmetrische Bedienlogik zum Schutz der Anlage implementiert.

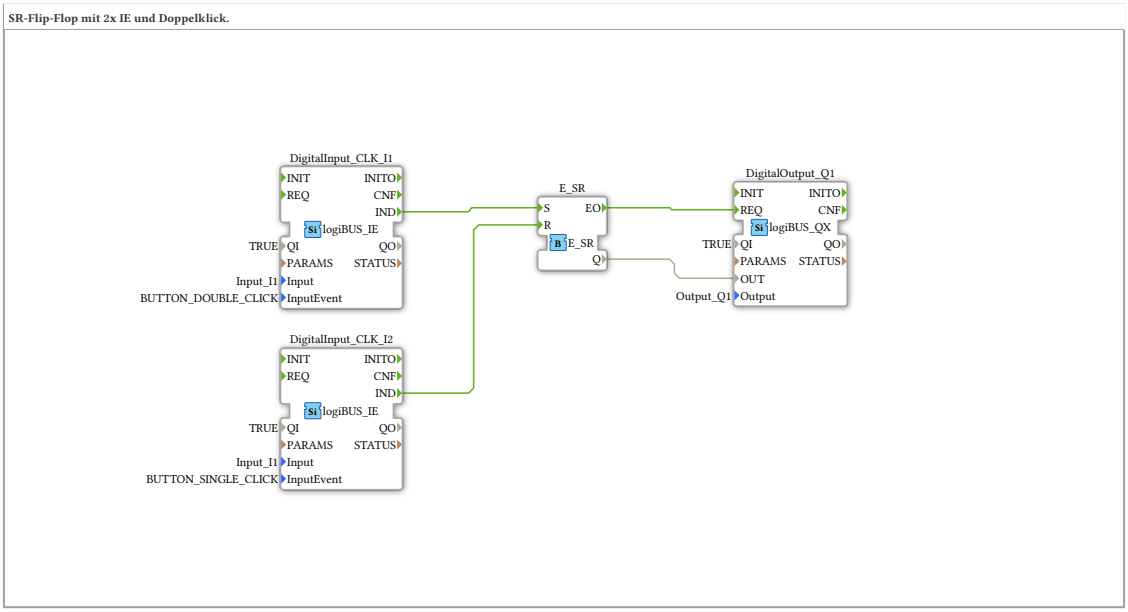
Ziel der Übung

Kombination von komplexen Eingangsereignissen (Doppelklick) mit Speicherbausteinen.

Beschreibung und Komponenten

Die Subapplikation `Uebung_006d.SUB` realisiert eine Ein/Aus-Logik mit unterschiedlichen Hürden.

Funktionsbausteine (FBs)



- **`I1` (Set)**: Konfiguriert auf `BUTTON_DOUBLE_CLICK`.

- **I2 (Reset)**: Konfiguriert auf `BUTTON_SINGLE_CLICK`.
- **E_SR**: Der Speicherbaustein.

Funktionsweise

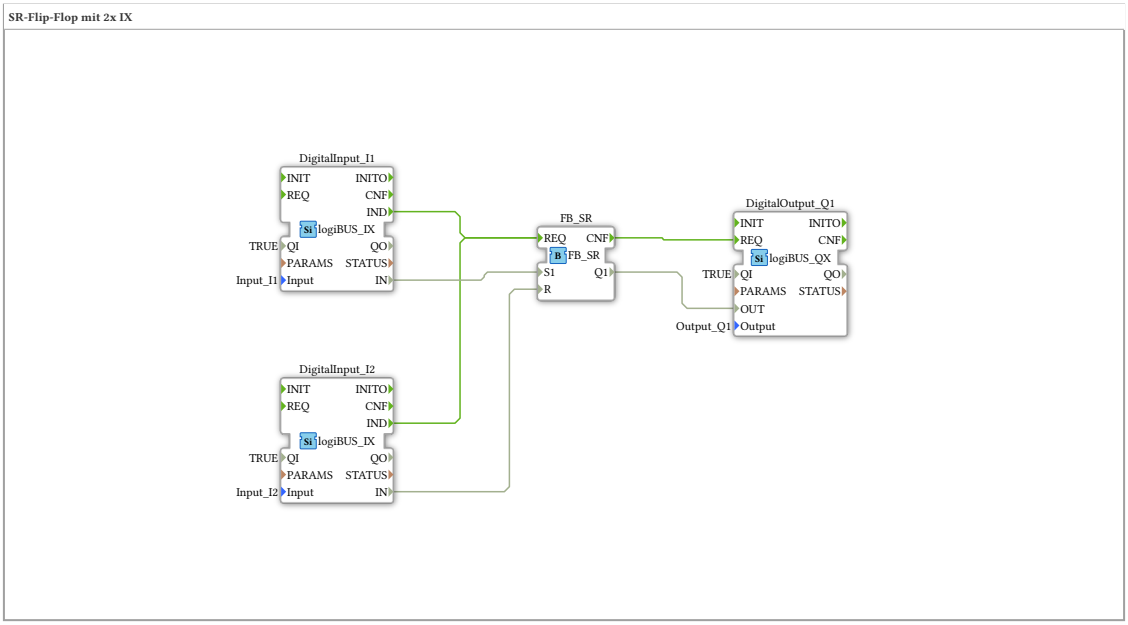
- **Einschalten**: Erfordert eine bewusste Handlung des Nutzers (Doppelklick auf `I1`). Ein einfaches Berühren reicht nicht aus.
- **Ausschalten**: Muss im Bedarfsfall schnell und einfach gehen (einfacher Klick auf `I2`).

Das Flip-Flop speichert den Zustand zwischen diesen Ereignissen.

Anwendungsbeispiel

Sicherheitsrelevante Hilfsantriebe: Eine hydraulische Pumpe oder ein Schneidwerk soll nicht durch ein versehentliches Anstoßen des Schalters in der Kabine starten. Der Nutzer muss durch den Doppelklick seine Absicht bestätigen. Das sofortige Ausschalten bei Gefahr muss jedoch durch einen einfachen Schlag auf den Aus-Taster gewährleistet sein.

2.2.1.86 Uebung_006e1: SR-Flip-Flop mit 2x IX



• – ○ • – ○ • – ○ •

Einleitung

Die Übung **Uebung_006e1** demonstriert die Implementierung eines SR-Flip-Flops (Set-Reset-Speicherglied) innerhalb einer Sub-Application. Es werden zwei digitale Eingänge verwendet, um einen digitalen Ausgang zu steuern. Diese Schaltung veranschaulicht das grundlegende Speicherverhalten in der Steuerungstechnik, bei dem ein Impuls einen Zustand setzt, der bis zum Rücksetzen erhalten bleibt.

Verwendete Funktionsbausteine (FBs)

In dieser Übung wird ein Netzwerk aus Standard-Bausteinen verwendet, um die gewünschte Logik abzubilden. Da es sich hierbei um eine `SubAppType` handelt, werden die darin enthaltenen Instanzen als interne FBs beschrieben.

Sub-Bausteine: Uebung_006e1 (Netzwerk)

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **DigitalInput_I1**: `logiBUS::io::DI::logiBUS_IX`
    - Parameter: `QI` = `TRUE`
    - Parameter: `Input` = `Input_I1`

    - Ereignisausgang: `IND` (Indication - Signaländerung)
    - Datenausgang: `IN` (Aktueller Wert des Eingangs)
  - **DigitalInput_I2**: `logiBUS::io::DI::logiBUS_IX`
    - Parameter: `QI` = `TRUE`
    - Parameter: `Input` = `Input_I2`
    - Ereignisausgang: `IND` (Indication - Signaländerung)
    - Datenausgang: `IN` (Aktueller Wert des Eingangs)
  - **DigitalOutput_Q1**: `logiBUS::io::DQ::logiBUS_QX`
    - Parameter: `QI` = `TRUE`
    - Parameter: `Output` = `Output_Q1`
    - Ereigniseingang: `REQ` (Request - Aktualisierung anfordern)
    - Dateneingang: `OUT` (Zu schreibender Wert)
  - **FB_SR**: `iec61131::bistableElements::FB_SR`
    - Ereigniseingang: `REQ`
    - Ereignisausgang: `CNF`
    - Dateneingang: `S1` (Setzen)
    - Dateneingang: `R` (Rücksetzen)
    - Datenausgang: `Q1` (Speicherzustand)
- **Funktionsweise**: Die SubApp liest zwei Hardware-Eingänge ein. Der erste Eingang dient als „Set"-Signal, der zweite als „Reset"-Signal für einen SR-Speicherbaustein. Der resultierende Zustand wird auf einen Hardware-Ausgang geschrieben.

Programmablauf und Verbindungen

Das Netzwerk verknüpft die physischen Ein- und Ausgänge mit der logischen SR-Funktion:

1. **Eingangsverarbeitung**:
   - `DigitalInput_I1` ist mit dem Eingang `S1` (Setzen) des `FB_SR` verbunden.
   - `DigitalInput_I2` ist mit dem Eingang `R` (Rücksetzen) des `FB_SR` verbunden.
   - Sobald sich an einem der Eingänge der Wert ändert (Event `IND`), wird die Berechnung im `FB_SR` über das Event `REQ` angestoßen.
2. **Logik (SR-Flip-Flop)**:
   - Der Baustein `FB_SR` speichert den Zustand.
   - Ist `S1` TRUE, wird der Ausgang `Q1` auf TRUE gesetzt.
   - Ist `R` TRUE, wird der Ausgang `Q1` auf FALSE gesetzt.
   - (Hinweis: Bei SR-Gliedern ist in der Regel das Setzen dominant, wenn beide Eingänge gleichzeitig TRUE sind, hängt dies jedoch von der spezifischen Implementierung der IEC 61131 Library ab; typischerweise ist ein SR-Glied Rücksetz-Dominant, wenn es als SR bezeichnet wird, aber die IEC Norm definiert SR als Set-Dominant. In 4diac/IEC61499 ist `FB_SR` definiert: Wenn S1 und R beide 1 sind, ist Q1=1).
3. **Ausgangsverarbeitung**:
   - Das Ergebnis `Q1` des Flip-Flops wird an den Dateneingang `OUT` von `DigitalOutput_Q1` geleitet.
   - Nach erfolgter Berechnung im Flip-Flop (Event `CNF`) wird der Ausgangsbaustein getriggert (`REQ`), um den physischen Ausgang zu aktualisieren.

Lernziele:

- Verständnis von bistabilen Kippstufen (Flip-Flops).
- Unterscheidung zwischen Setzen und Rücksetzen.
- Verknüpfung von Event- und Datenflüssen zwischen IO-Bausteinen und Logik-Bausteinen.

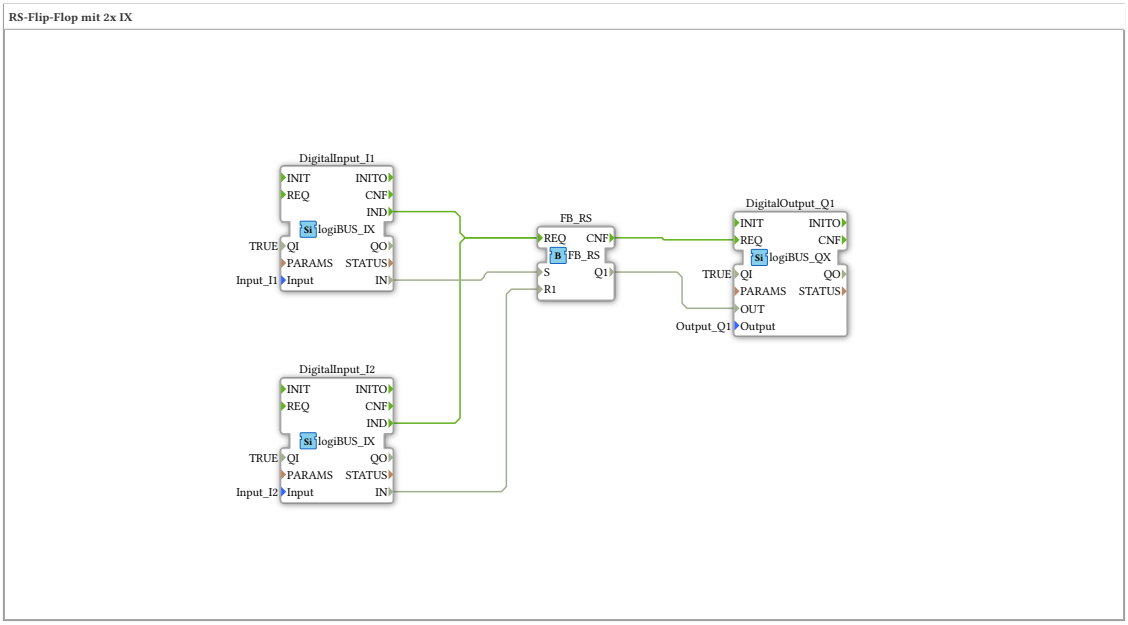
Zusammenfassung

Die Übung `Uebung_006e1` ist eine klassische Anwendung einer Speicherfunktion. Mit Hilfe von zwei Tastern (oder Schaltern) an den Eingängen `Input_I1` und `Input_I2` kann der Ausgang `Output_Q1` dauerhaft ein- bzw. ausgeschaltet werden. Dies bildet die Grundlage für viele Steuerungsaufgaben, wie z.B. Start/Stopp-Schaltungen für Motoren.

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

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

2.2.1.87 Uebung_006e2: RS-Flip-Flop mit 2x IX



• – ○ • – ○ • – ○ •

Einleitung

Die Übung **Uebung_006e2** demonstriert die Implementierung eines **RS-Flip-Flops** (Rücksetz-Dominant) unter Verwendung von zwei digitalen Eingängen (IX) und einem digitalen Ausgang. Das Ziel ist es, eine grundlegende Speicherfunktion zu realisieren, bei der ein Eingang den Ausgang setzt und der andere ihn zurücksetzt. Diese Übung nutzt die `logiBUS` Bibliothek für die Hardware-Abstraktion der Ein- und Ausgänge.

Verwendete Funktionsbausteine (FBs)

In dieser Sub-Applikation werden verschiedene Bausteine instanziiert und miteinander verschaltet.

Sub-Bausteine: DigitalInput_I1

- **Typ**: `logiBUS::io::DI::logiBUS_IX`
- **Beschreibung**: Dieser Baustein repräsentiert den ersten digitalen Eingang, der als „Set“-Signal fungiert.
- **Parameter**:
  - `QI` = `TRUE` (Initialisierung aktiviert)
  - `Input` = `Input_I1` (Hardware-Mapping auf Eingang I1)

Sub-Bausteine: DigitalInput_I2

- **Typ**: `logiBUS::io::DI::logiBUS_IX`
- **Beschreibung**: Dieser Baustein repräsentiert den zweiten digitalen Eingang, der als „Reset“-Signal fungiert.
- **Parameter**:
  - `QI` = `TRUE`
  - `Input` = `Input_I2` (Hardware-Mapping auf Eingang I2)

Sub-Bausteine: FB_RS

- **Typ**: `iec61131::bistableElements::FB_RS`
- **Beschreibung**: Ein bistabiles Element (Flip-Flop) mit Rücksetz-Dominanz.
- **Funktionsweise**:
  - Wenn der Eingang `S` (Set) TRUE ist und `R1` (Reset) FALSE ist, wird der Ausgang `Q1` TRUE.

- Wenn der Eingang `R1` TRUE ist, wird der Ausgang `Q1` FALSE (unabhängig von S, da RS rücksetzdominant ist).

Sub-Bausteine: DigitalOutput_Q1

- **Typ**: `logiBUS::io::DQ::logiBUS_QX`
- **Beschreibung**: Dieser Baustein repräsentiert den digitalen Ausgang, der den Status des Flip-Flops anzeigt.
- **Parameter**:
  - `QI` = `TRUE`
  - `Output` = `Output_Q1` (Hardware-Mapping auf Ausgang Q1)

Programmablauf und Verbindungen

Der Programmablauf wird durch die Ereignisketten (Event Connections) und den Datenfluss (Data Connections) bestimmt:

1. **Eingangsverarbeitung**:
   - Die digitalen Eingänge `DigitalInput_I1` und `DigitalInput_I2` erfassen Signale von der Hardware.
   - Sobald sich ein Eingangswert ändert oder aktualisiert wird, wird ein `IND`-Event (Indication) ausgelöst.
2. **Logikverarbeitung (RS-Flip-Flop)**:
   - Die `IND`-Events beider Eingänge sind mit dem `REQ`-Eingang (Request) des `FB_RS` verbunden. Das bedeutet, jede Änderung an I1 oder I2 triggert die Berechnung des Flip-Flops.
   - **Datenverbindung**:
     - Der Wert von `DigitalInput_I1` (`IN`) ist mit dem Set-Eingang (`S`) des `FB_RS` verbunden.
     - Der Wert von `DigitalInput_I2` (`IN`) ist mit dem Reset-Eingang (`R1`) des `FB_RS` verbunden.
3. **Ausgangsverarbeitung**:
   - Nach der Berechnung des `FB_RS` wird das `CNF`-Event (Confirmation) ausgelöst.
   - Dieses Event ist mit dem `REQ`-Eingang des `DigitalOutput_Q1` verbunden, um den Ausgang zu aktualisieren.
   - **Datenverbindung**: Der Ergebnisausgang `Q1` des Flip-Flops wird an den Eingang `OUT` des `DigitalOutput_Q1` übergeben.

Zusammengefasstes Verhalten:

- Taste/Schalter an **Input_I1** aktiviert den Ausgang **Output_Q1**.
- Taste/Schalter an **Input_I2** deaktiviert den Ausgang **Output_Q1**.
- Werden beide Eingänge gleichzeitig betätigt, bleibt der Ausgang aus (Reset ist dominant).

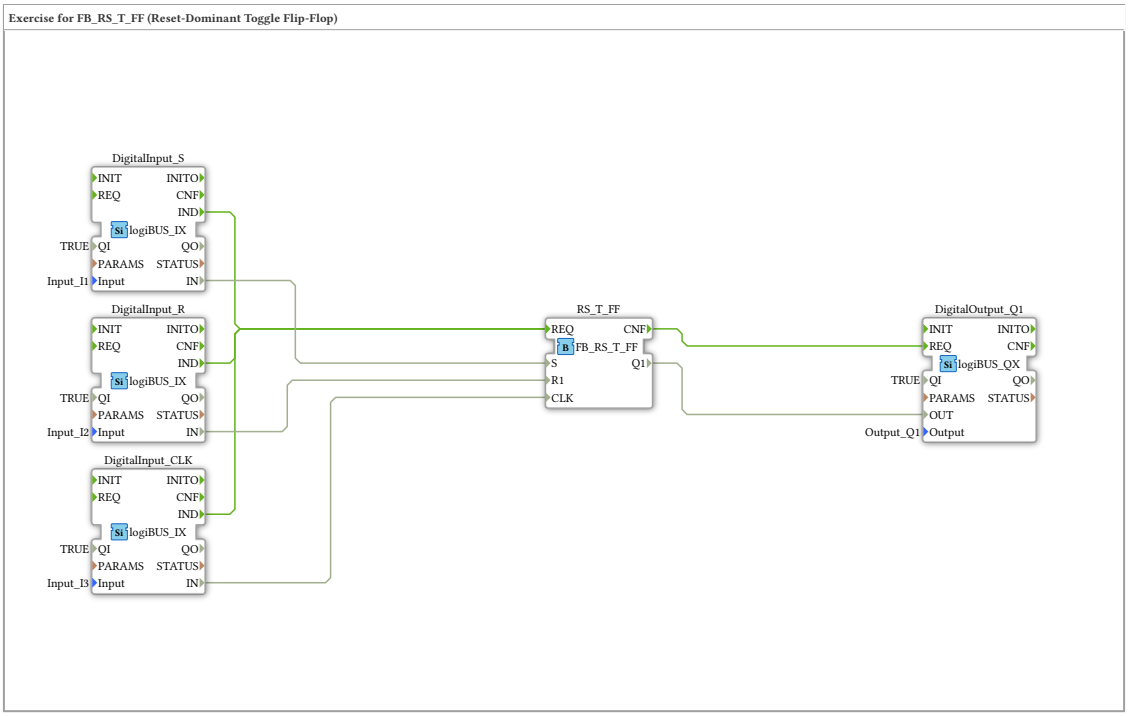
Zusammenfassung

Diese Übung ist ein klassisches Beispiel für speicherprogrammierbare Steuerungslogik nach IEC 61131-3. Sie vermittelt das Verständnis für bistabile Elemente und die Ereignissteuerung in 4diac, bei der die Ausführung der Logikblöcke durch Trigger (Events) von den Eingangsbausteinen gesteuert wird. Das Ergebnis ist eine robuste Schaltung zum Ein- und Ausschalten eines Verbrauchers über zwei getrennte Signale.

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

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

2.2.1.88 Uebung_006f_RS: Exercise for FB_RS_T_FF (Reset-Dominant Toggle Flip-Flop)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert die Verwendung eines **reset-dominanten Toggle-Flipflops** (FB_RS_T_FF) in der 4diac-IDE. Ziel ist es, das Verhalten eines priorisierten Rücksetzeingangs in Kombination mit einem taktgesteuerten Toggle-Mechanismus zu verstehen. Die Eingänge werden über logiBUS-Hardwarekomponenten angeschlossen, die Ausgabe erfolgt über einen digitalen Ausgang.

Verwendete Funktionsbausteine (FBs)

Die Übung besteht aus fünf Funktionsbausteinen, die direkt im SubApp-Netzwerk verbunden sind:

Bausteinname	Typ	Zweck
`DigitalInput_S`	logiBUS::io::DI::logiBUS_IX	Digitaler Eingang für den Set-Eingang (S)
`DigitalInput_R`	logiBUS::io::DI::logiBUS_IX	Digitaler Eingang für den Reset-Eingang (R1)
`DigitalInput_CLK`	logiBUS::io::DI::logiBUS_IX	Digitaler Eingang für den Takteingang (CLK)
`RS_T_FF`	logiBUS::bistable-Elements::FB_RS_T_FF	Reset-dominantes Toggle-Flipflop
`DigitalOutput_Q1`	logiBUS::io::DQ::logiBUS_QX	Digitaler Ausgang für das Flipflop-Signal (Q1)

Details zu den Bausteinen

`DigitalInput_S`, `DigitalInput_R`, `DigitalInput_CLK`

- **Typ**: logiBUS::io::DI::logiBUS_IX (Hardware-Eingangsbaustein)
- **Parameter**:
  - `QI` = `TRUE` (Eingang aktiviert)
  - `Input` = `Input_I1`, `Input_I2`, `Input_I3` (physische Eingänge)
- **Ereignisausgang**: `IND` (wird ausgelöst, wenn sich der Eingangswert ändert)

- **Datenausgang**: `IN` (aktueller digitaler Wert des Eingangs)

`RS_T_FF`

- **Typ**: logiBUS::bistableElements::FB_RS_T_FF
- **Parameter**: Keine benutzerdefinierten Parameter
- **Ereigniseingang**: `REQ` (startet die Verarbeitung)
  - Verbunden mit den `IND`-Ereignissen aller drei Eingangsbausteine
- **Ereignisausgang**: `CNF` (zeigt Abschluss der Verarbeitung an)
- **Daten-/Eingänge**:
  - `S` (Set, von `DigitalInput_S.IN`)
  - `R1` (Reset, von `DigitalInput_R.IN`)
  - `CLK` (Takt, von `DigitalInput_CLK.IN`)
- **Datenausgang**: `Q1` (Ausgangszustand des Flipflops)

`DigitalOutput_Q1`

- **Typ**: logiBUS::io::DQ::logiBUS_QX (Hardware-Ausgangsbaustein)
- **Parameter**:
  - `QI` = `TRUE` (Ausgang aktiviert)
  - `Output` = `Output_Q1` (physischer Ausgang)
- **Ereigniseingang**: `REQ` (vom `CNF` des Flipflops)
- **Dateneingang**: `OUT` (vom `Q1` des Flipflops)

Programmablauf und Verbindungen

Der Programmablauf wird durch die physischen Eingänge I1, I2 und I3 gesteuert. Jede Änderung an einem dieser Eingänge führt zu folgenden Schritten:

1. **Ereignisauslösung**: Der betroffene Eingangsbaustein (`DigitalInput_S`, `DigitalInput_R` oder `DigitalInput_CLK`) sendet ein `IND`-Ereignis.
2. **Ereignisweiterleitung**: Alle drei `IND`-Ereignisse sind mit dem `REQ`-Eingang des Flipflops `RS_T_FF` verbunden. Somit wird das Flipflop bei **jeder** Änderung eines Eingangs neu berechnet.
3. **Datenverarbeitung im Flipflop**:
   - Der Set-Eingang `S` wird durch `DigitalInput_S.IN` (I1) gespeist.
   - Der Reset-Eingang `R1` wird durch `DigitalInput_R.IN` (I2) gespeist.
   - Der Takteingang `CLK` wird durch `DigitalInput_CLK.IN` (I3) gespeist.
   - Das Flipflop verhält sich reset-dominant: Wenn `R1 = TRUE`, wird der Ausgang `Q1` unabhängig von `S` und der Taktflanke zurückgesetzt. Bei `R1 = FALSE` und positiver Taktflanke auf `CLK` wird der Ausgang getoggelt, wenn `S = TRUE`.
4. **Ausgabe**: Nach der Verarbeitung sendet `RS_T_FF` ein `CNF`-Ereignis an den Ausgangsbaustein `DigitalOutput_Q1`. Der neue Zustand `Q1` wird auf den physischen Ausgang Q1 übertragen.

Wichtige Hinweise:

- Durch die Ereignisverbindung aller drei `IND`-Ereignisse auf den gleichen `REQ`-Eingang wird das Flipflop bei jeder Eingangsänderung neu berechnet, auch wenn der Auslöser kein Taktwechsel ist. Dies kann zu unerwartetem Verhalten führen, wenn die Logik nicht berücksichtigt, dass das Flipflop eigentlich taktgesteuert arbeitet. In der Praxis sollte der Takt separat behandelt werden.
- Die Übung dient dem Verständnis der Priorisierung von Reset vor Toggle in einem reset-dominanten Flipflop.

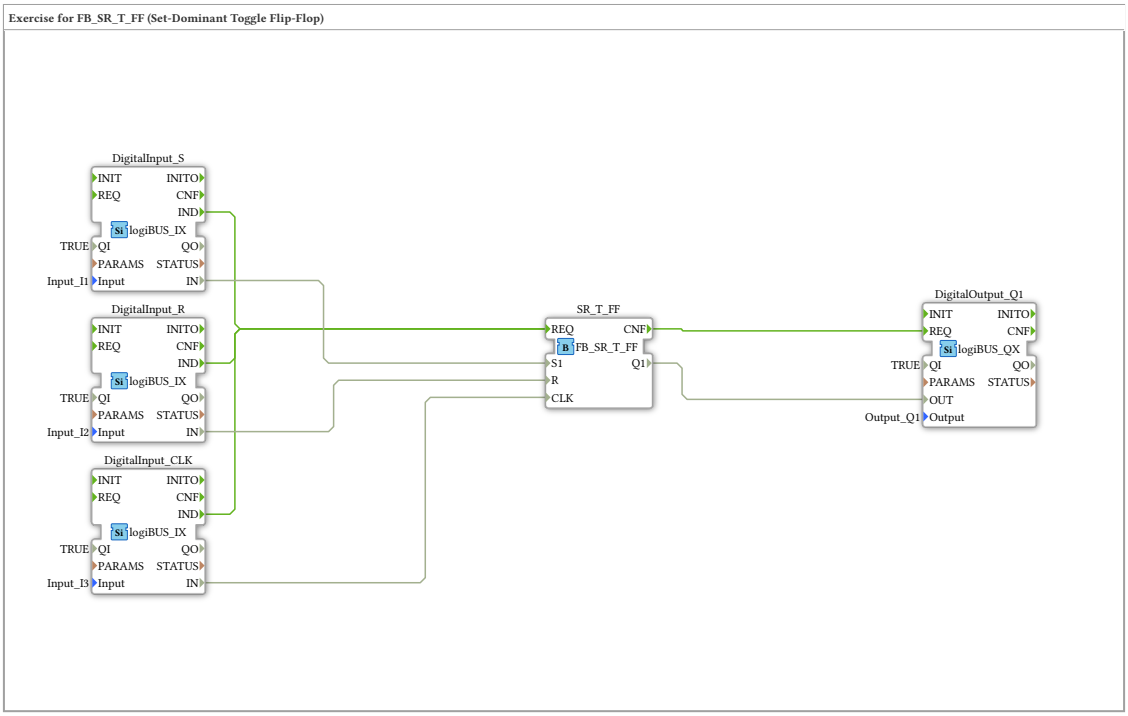
Zusammenfassung

Die Übung `Uebung_006f_RS` zeigt die Einbindung eines reset-dominanten Toggle-Flipflops in eine 4diac-Anwendung mit logiBUS-Hardware. Die drei Eingänge (Set, Reset, Takt) werden über digitale Eingangsbausteine eingelesen. Der Ausgang des Flipflops steuert einen digitalen Ausgang. Der Ablauf verdeutlicht die Verarbeitung von Ereignisketten und die Funktion eines priorisierten Resets. Als Lernziel steht das Verständnis der Toggle-Logik und der Dominanz des Reset-Signals im Vordergrund.

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

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

2.2.1.89 Uebung_006f_SR: Exercise for FB_SR_T_FF (Set-Dominant Toggle Flip-Flop)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung dient dem Verständnis und der Anwendung eines **set-dominanten Toggle-Flipflops** (FB_SR_T_FF). Der Baustein kombiniert die Eigenschaften eines SR-Flipflops mit einer Toggle-Funktion, wobei der Set-Eingang (S1) Vorrang vor dem Reset-Eingang (R) hat. Die Übung demonstriert die grundlegende Verschaltung von digitalen Eingängen, dem Flipflop und einem digitalen Ausgang in der 4diac-IDE unter Verwendung der logiBUS-Bibliothek.

Verwendete Funktionsbausteine (FBs)

Die Übung besteht aus fünf Funktionsbausteinen, die im logiBUS-Netzwerk miteinander verbunden sind.

- **DigitalInput_S** (Typ: *logiBUS::io::DI::logiBUS_IX*)
  - **Parameter**: QI = TRUE, Input = Input_I1
  - **Ereignisausgang**: IND (wird bei anliegendem Eingangssignal ausgelöst)
  - **Datenausgang**: IN (logischer Wert des physischen Eingangs)
  - **Funktionsweise**: Liest den digitalen Eingang I1 (z. B. Taster oder Sensor) und stellt den Wert als Datensignal sowie das Ereignis IND bereit.
- **DigitalInput_R** (Typ: *logiBUS::io::DI::logiBUS_IX*)
  - **Parameter**: QI = TRUE, Input = Input_I2
  - **Ereignisausgang**: IND
  - **Datenausgang**: IN
  - **Funktionsweise**: Liest den digitalen Eingang I2 (Reset-Taster) und gibt den Wert sowie das Ereignis aus.
- **DigitalInput_CLK** (Typ: *logiBUS::io::DI::logiBUS_IX*)
  - **Parameter**: QI = TRUE, Input = Input_I3
  - **Ereignisausgang**: IND
  - **Datenausgang**: IN
  - **Funktionsweise**: Liest den digitalen Eingang I3 (Taktsignal) und stellt den Wert sowie das Ereignis bereit.
- **SR_T_FF** (Typ: *logiBUS::bistableElements::FB_SR_T_FF*)

  - **Parameter**: Keine (werkseitige Konfiguration)
  - **Ereigniseingang**: REQ (löst die Verarbeitung aus)
  - **Ereignisausgang**: CNF (bestätigt die Ausführung)
  - **Dateneingänge**: S1 (Set, dominant), R (Reset), CLK (Takt)
  - **Datenausgang**: Q1 (Ausgangszustand)
  - **Funktionsweise**: Realisiert ein set-dominantes Toggle-Flipflop. Bei jedem Taktflanke (CLK) wird der Ausgang Q1 gesetzt, wenn S1 aktiv ist, oder zurückgesetzt, wenn R aktiv ist. Sind beide Eingänge aktiv, dominiert S1 (Set). Der Ausgang toggelt, wenn weder S1 noch R aktiv sind, jedoch nur bei steigender Taktflanke.
- **DigitalOutput_Q1** (Typ: *logiBUS::io::DQ::logiBUS_QX*)
  - **Parameter**: QI = TRUE, Output = Output_Q1
  - **Ereigniseingang**: REQ (löst das Setzen des Ausgangs aus)
  - **Dateneingang**: OUT (Wert, der auf den physischen Ausgang gegeben wird)
  - **Funktionsweise**: Gibt den übergebenen Wert (OUT) auf den digitalen Ausgang Q1 (z. B. LED oder Aktor) aus.

Programmablauf und Verbindungen

Die folgenden Ereignis- und Datenverbindungen definieren den Ablauf der Übung:

Ereignisverbindungen:

- Die Ereignisausgänge der drei digitalen Eingänge (DigitalInput_S.IND, DigitalInput_R.IND, DigitalInput_CLK.IND) sind alle mit dem Ereigniseingang des Flipflops (SR_T_FF.REQ) verbunden. *Hinweis:* Dies bedeutet, dass jede Änderung an einem der Eingänge (S, R oder CLK) die Verarbeitung des Flipflops auslöst. In der Praxis sollte der Takt (CLK) den Hauptauslöser darstellen; die gleichzeitige Verknüpfung aller drei Eingänge ist hier als vereinfachte Übung gewählt.
- Das Bestätigungsereignis des Flipflops (SR_T_FF.CNF) ist mit dem Ereigniseingang des Ausgangsbausteins (DigitalOutput_Q1.REQ) verbunden, sodass nach jeder Flipflop-Berechnung der Ausgang aktualisiert wird.

Datenverbindungen:

- DigitalInput_S.IN → SR_T_FF.S1 (Set-Eingang)
- DigitalInput_R.IN → SR_T_FF.R (Reset-Eingang)
- DigitalInput_CLK.IN → SR_T_FF.CLK (Taktsignal)
- SR_T_FF.Q1 → DigitalOutput_Q1.OUT (Ausgangswert)

Ablauf:

1. Ein Signal an einem der digitalen Eingänge (I1, I2 oder I3) erzeugt ein Ereignis (IND).
2. Dieses Ereignis triggert den Flipflop-Baustein (REQ).
3. Der Flipflop wertet die aktuellen Datenwerte an S1, R und CLK aus und berechnet den neuen Zustand Q1 gemäß der set-dominanten Toggle-Logik.
4. Nach der Berechnung signalisiert der Flipflop die Fertigstellung (CNF).
5. Der Ausgangsbaustein übernimmt den Wert (OUT) und setzt den physischen Ausgang Q1 entsprechend.

Lernziele:

- Verständnis der Funktionsweise eines set-dominanten Toggle-Flipflops (SR_T_FF).
- Umgang mit logiBUS-Eingangs- und Ausgangsbausteinen.
- Verknüpfung von Ereignis- und Datenflüssen in der 4diac-IDE.
- Praktische Anwendung der logiBUS-Bibliothek zur Steuerung von Hardware.

Schwierigkeitsgrad: Einfach bis mittel. Grundkenntnisse in der 4diac-IDE und im Umgang mit logiBUS-Komponenten werden vorausgesetzt.

Vorkenntnisse: Grundlagen der binären Logik und Funktionsweise von Flipflops.

Durchführung: Die Übung kann in der 4diac-IDE geladen und auf einer entsprechenden Hardware (z. B. Siemens SPS mit logiBUS) oder im Simulationsmodus ausgeführt werden. Die Eingänge I1, I2 und I3 müssen physisch mit Tastern oder Signalquellen verbunden sein. Der Ausgang Q1 wird z. B. an eine LED angeschlossen.

Zusammenfassung

Die Übung *Uebung_006f_SR* vermittelt die praktische Anwendung eines set-dominanten Toggle-Flipflops (FB_SR_T_FF) in der 4diac-IDE. Durch die Verschaltung von drei digitalen Eingängen, dem Flipflop und einem digitalen Ausgang wird gezeigt, wie Ereignis- und Datenflüsse gesteuert werden können. Der Baustein verhält sich set-dominant: Bei gleichzeitig anliegenden Set- und Reset-Signalen hat der Set-Eingang Vorrang. Diese Übung ist ein grundlegender Baustein für das Verständnis sequenzieller Logik in der industriellen Automatisierungstechnik.

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

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

2.2.1.90 Uebung_007: Blinker mit E_CYCLE und E_T_FF

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_007`. Hier wird gezeigt, wie man periodische Ereignisse erzeugt, um ein zyklisches Blinksignal zu realisieren.

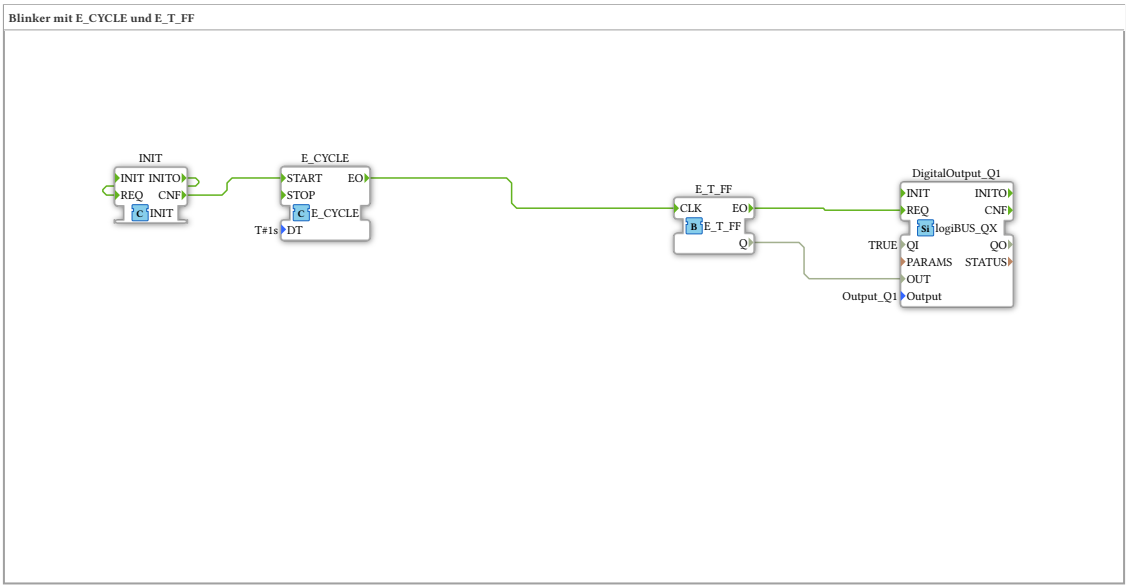
Ziel der Übung

Verwendung des `E_CYCLE` Bausteins zur Erzeugung einer Zeitbasis. Es wird demonstriert, wie ein periodischer Trigger ein Toggle-Flip-Flop ansteuert, um ein gleichmäßiges Rechtecksignal (An/Aus) zu generieren.

Beschreibung und Komponenten

Die Subapplikation `Uebung_007.SUB` kombiniert einen Taktgeber mit einem Speicherglied.

Funktionsbausteine (FBs)



- **`E_CYCLE`**: Ein Ereignis-Generator. Er sendet periodisch Ereignisse am Ausgang `EO` aus. Der Parameter `DT` bestimmt das Zeitintervall (hier `T#1s` = 1 Sekunde).
- **`E_T_FF`**: Das Toggle-Flip-Flop, welches bei jedem Takt seinen Zustand invertiert.
- **`DigitalOutput_Q1`**: Die physische Lampe.

Funktionsweise

1. Der `E_CYCLE` Baustein feuert jede Sekunde ein Ereignis ab.
2. Dieses Ereignis erreicht den `CLK`-Eingang des `E_T_FF`.

3. Das Flip-Flop wechselt bei jedem Impuls den Zustand (Aus ➡️ An ➡️ Aus ➡️ ...).
4. Da für einen vollen Zyklus (An und Aus) zwei Taktimpulse benötigt werden, blinkt die Lampe mit einer Frequenz von 0,5 Hz (1 Sekunde an, 1 Sekunde aus).

Anwendungsbeispiel

Betriebsanzeige: Eine grüne LED am Schaltschrank blinkt langsam, um zu signalisieren, dass die Steuerung aktiv ist und das Programm ordnungsgemäß ausgeführt wird.

2.2.1.91 Uebung_007a1: Blinker mit E_CYCLE und E_T_FF



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_007a1`. Hier wird versucht, den Blinker aus Übung 007 über externe Taster ein- und auszuschalten.

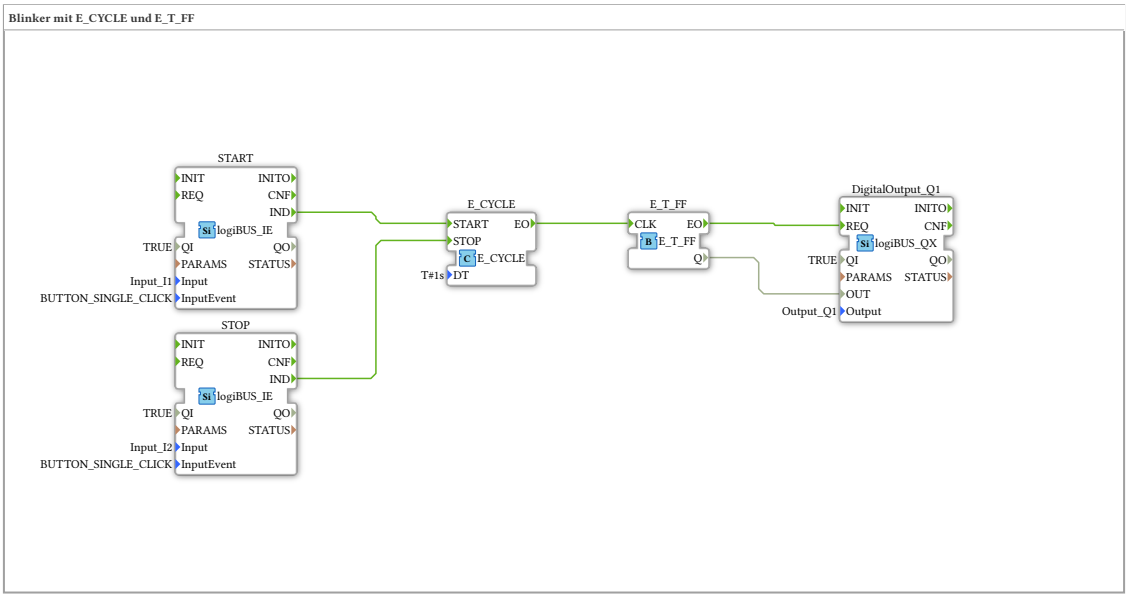
Ziel der Übung

Steuerung eines Taktgebers über Start- und Stopp-Ereignisse.

Beschreibung und Komponenten

In `Uebung_007a1.SUB` werden die Steuerungseingänge des `E_CYCLE` Bausteins genutzt.

Funktionsbausteine (FBs)



- **`START` (I1)**: Sendet ein Ereignis an `E_CYCLE.START`.
- **`STOP` (I2)**: Sendet ein Ereignis an `E_CYCLE.STOP`.
- **`E_CYCLE`**: Startet oder stoppt die Generierung von Takt-Events.

Das Problem

Wie im Kommentar der Übung vermerkt: *„dieser Blinker bleibt zufällig auf AN oder AUS stehen“*. Wenn der `STOP`-Befehl eintrifft, stellt der `E_CYCLE` sofort seine Arbeit ein. Das nachgeschaltete Flip-Flop `E_T_FF` verharrt jedoch in seinem **letzten Zustand**. War die Lampe in diesem Moment gerade an, bleibt sie dauerhaft leuchten. Dies ist in der Automatisierungstechnik meist unerwünscht und potenziell gefährlich.

Fazit

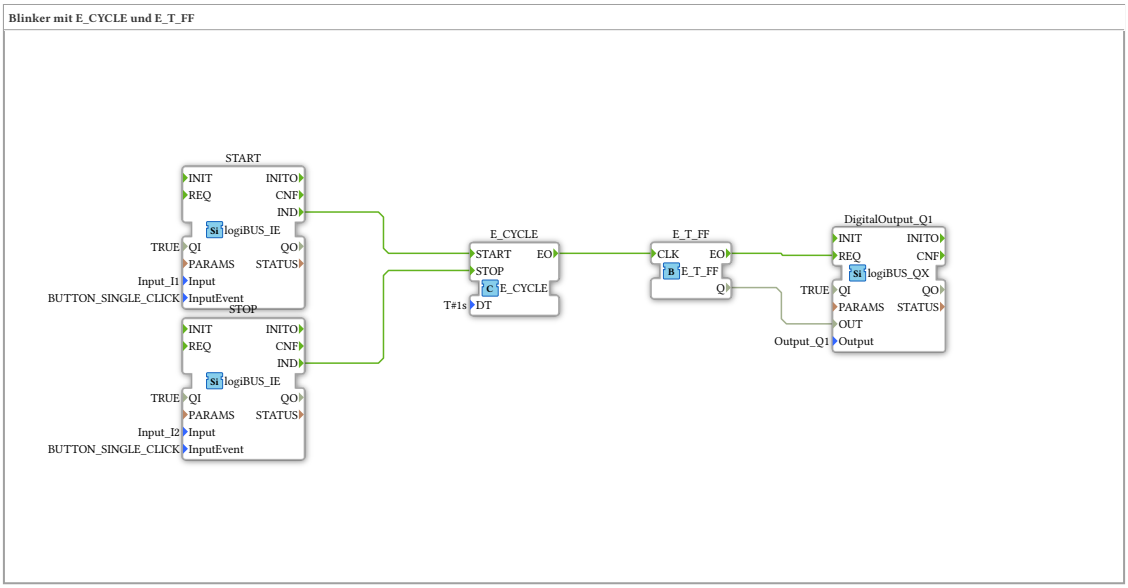
Diese Übung dient als Lehrbeispiel dafür, dass das bloße Anhalten eines Taktgebers nicht ausreicht, um ein System in einen sicheren (ausgeschalteten) Zustand zu überführen.

2.2.1.92 Uebung_007a2: Blinker mit E_CYCLE und E_T_FF

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_007a2`.

Übersicht

Strukturell ist diese Übung identisch mit `Uebung_007a1` und dient der Festigung des Wissens. Auch hier besteht das Problem des undefinierten Endzustands beim Stoppen des Blinkers.



2.2.1.93 Uebung_007a3: Blinker mit E_CYCLE, E_SWITCH und E_SR

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_007a3`. Hier wird die „saubere“ Lösung für einen schaltbaren Blinker präsentiert, der beim Ausschalten garantiert in den Zustand „AUS“ geht.

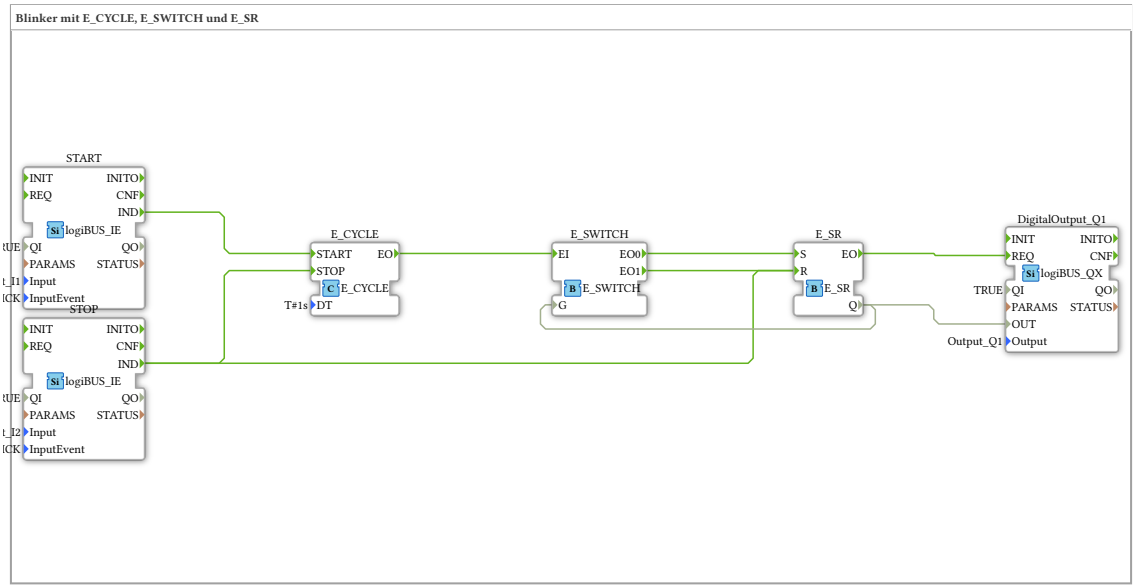
Ziel der Übung

Realisierung eines Blinkers mit definiertem Stopp-Verhalten. Es wird demonstriert, wie Ereignispfade verknüpft werden müssen, um sowohl die Takterzeugung zu stoppen als auch den Zustandsspeicher zu löschen.

Beschreibung und Komponenten

In `Uebung_007a3.SUB` wird die Blinklogik manuell aus Weiche und Speicher aufgebaut, um die volle Kontrolle über den Reset-Vorgang zu haben.

Funktionsbausteine (FBs)



- **`E_CYCLE`**: Der Taktgeber.
- **`E_SWITCH`**: Die Ereignis-Weiche zur Realisierung der Toggle-Funktion.
- **`E_SR`**: Der Speicherbaustein (Reset-dominant verschaltet).
- **`START` & `STOP`**: Die Bedien-Taster.

Funktionsweise

Die Sicherheit wird durch eine doppelte Belegung des Stopp-Signals erreicht:

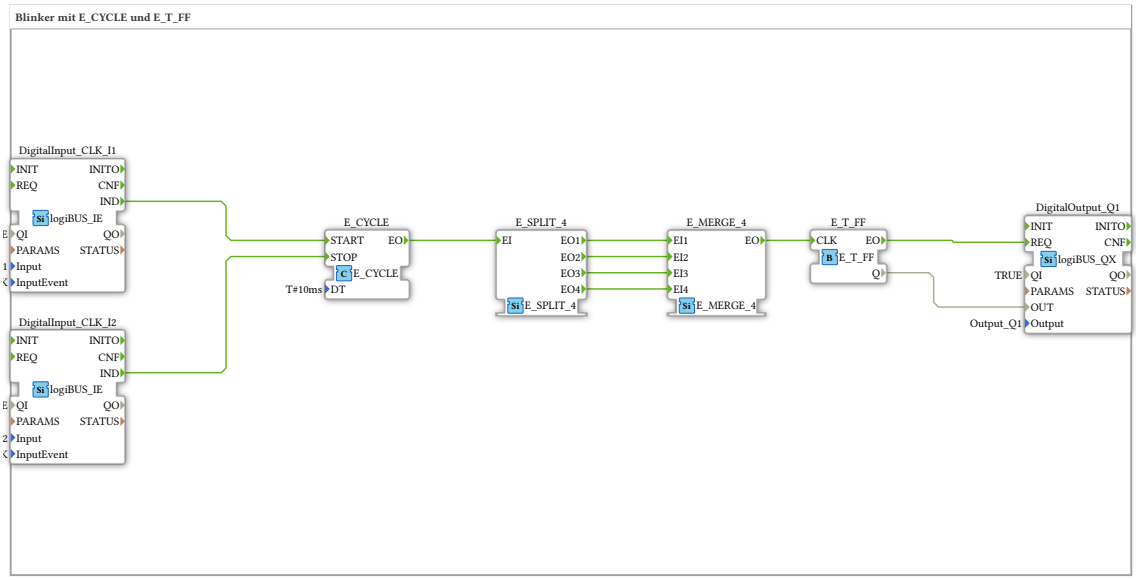
```
<EventConnections>
    <Connection Source="START.IND" Destination="E_CYCLE.START"/>
    <Connection Source="STOP.IND" Destination="E_CYCLE.STOP"/>
    <Connection Source="E_CYCLE.EO" Destination="E_SWITCH.EI"/>
    <Connection Source="E_SWITCH.EO0" Destination="E_SR.S"/>
    <Connection Source="E_SWITCH.EO1" Destination="E_SR.R"/>
    <!-- Die entscheidende Verbindung für die Sicherheit: -->
    <Connection Source="STOP.IND" Destination="E_SR.R"/>
</EventConnections>
```

1. **Blinkbetrieb**: Der `E_CYCLE` triggert die `E_SWITCH/E_SR` Kombination, was zum periodischen Umschalten führt.
2. **Ausschalten**: Wenn der Nutzer `STOP` drückt, passieren zwei Dinge gleichzeitig:
   - Der `E_CYCLE` wird angehalten (keine neuen Takte mehr).
   - Der Speicher `E_SR` erhält einen **direkten Reset-Befehl**. Damit wird der Ausgang sofort auf `FALSE` gezwungen, egal ob das Flip-Flop gerade im „An“- oder „Aus“-Zustand war.

Anwendungsbeispiel

Professionelle Warnsignalisierung: Ein akustischer Alarm oder eine Blitzleuchte an einer Maschine muss bei Quittierung sofort und zuverlässig verstummen. Ein „Hängenbleiben“ im eingeschalteten Zustand wäre irreführend und störend. Diese Schaltung garantiert, dass der Alarm nach dem Ausschalten immer inaktiv ist.

2.2.1.94 Uebung_007b: Blinker mit E_CYCLE und E_T_FF



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung realisiert einen einfachen Blinker, der durch zwei Taster gesteuert wird. Ein E_CYCLE-Funktionsbaustein erzeugt periodische Ereignisse, die über einen E_SPLIT_4 auf mehrere Pfade verteilt werden. Alle vier Ausgänge des Splitters werden in einem E_MERGE_4 wieder zusammengeführt, so dass jede Periode ein einzelnes Ereignis an den Toggle-Flipflop (E_T_FF) weitergegeben wird. Der Ausgang des Flipflops schaltet einen digitalen Ausgang (logiBUS Q1) um. Der Taktgenerator kann über einen Taster (I1) gestartet und über einen zweiten Taster (I2) gestoppt werden.

Die gesamte Logik ist in einer Subapplikation gekapselt und verwendet ausschließlich logiBUS-Hardware-Ein- und -Ausgänge.

Verwendete Funktionsbausteine (FBs)

Die Subapplikation besteht aus folgenden Funktionsbausteinen:

Baustein: `DigitalOutput_Q1`

- **Typ**: `logiBUS::io::DQ::logiBUS_QX`
- **Parameter**:
  - `QI` = `TRUE` (Freigabe)
  - `Output` = `Output_Q1` (Hardware-Ausgang)
- **Ereignis-Eingänge**: `REQ` (schaltet den Ausgang)
- **Daten-Eingänge**: `OUT` (Wert für den Ausgang, 0 oder 1)
- **Funktionsweise**: Stellt den logiBUS-Digitalausgang Q1 auf den Wert des Dateneingangs, sobald ein Ereignis an `REQ` eintrifft.

Baustein: `E_CYCLE`

- **Typ**: `iec61499::events::E_CYCLE`
- **Parameter**:
  - `DT` = `T#10ms` (Zykluszeit 10 ms)
- **Ereignis-Eingänge**:
  - `START` (startet die zyklische Erzeugung)
  - `STOP` (stoppt die zyklische Erzeugung)
- **Ereignis-Ausgänge**:
  - `EO` (gibt alle `DT` ein Ereignis aus)
- **Funktionsweise**: Generiert nach dem Starten periodisch Ereignisse im Abstand von 10 ms. Der Zähler kann durch ein Ereignis an `STOP` angehalten werden.

Baustein: `E_T_FF`

- **Typ**: `iec61499::events::E_T_FF`
- **Parameter**: Keine
- **Ereignis-Eingänge**:
  - `CLK` (Takt – jedes Ereignis toggelt den Ausgang)
- **Ereignis-Ausgänge**:
  - `EO` (wird bei jedem Takt ausgegeben)
- **Daten-Ausgänge**:
  - `Q` (aktueller Wert des Flipflops, 0 oder 1)
- **Funktionsweise**: Ein Toggle-Flipflop: bei jedem Ereignis an `CLK` wechselt der Ausgang `Q` zwischen 0 und 1. Gleichzeitig wird ein Ereignis an `EO` ausgelöst.

Baustein: `E_SPLIT_4`

- **Typ**: `iec61499::events::E_SPLIT_4`
- **Parameter**: Keine
- **Ereignis-Eingänge**: `EI` (Eingangsereignis)
- **Ereignis-Ausgänge**: `EO1`, `EO2`, `EO3`, `EO4` (vier parallele Ausgänge)
- **Funktionsweise**: Verteilt ein eingehendes Ereignis an alle vier Ausgänge gleichzeitig.

Baustein: `E_MERGE_4`

- **Typ**: `iec61499::events::E_MERGE_4`
- **Parameter**: Keine
- **Ereignis-Eingänge**: `EI1`, `EI2`, `EI3`, `EI4` (vier Eingänge)
- **Ereignis-Ausgänge**: `EO` (Ausgangsereignis)
- **Funktionsweise**: Sobald an einem der vier Eingänge ein Ereignis eintrifft, wird dieses sofort an den Ausgang weitergegeben. (Logische ODER-Verknüpfung der Ereignisse.)

Baustein: `DigitalInput_CLK_I1`

- **Typ**: `logiBUS::io::DI::logiBUS_IE`
- **Parameter**:
  - `QI` = `TRUE`
  - `Input` = `Input_I1` (Hardware-Eingang I1)
  - `InputEvent` = `BUTTON_SINGLE_CLICK` (Ereignis bei einfachem Tastendruck)
- **Ereignis-Ausgänge**: `IND` (wird bei erkanntem Ereignis ausgelöst)
- **Funktionsweise**: Erkennt einen einzelnen Tastendruck an Eingang I1 und gibt ein Ereignis an `IND` aus.

Baustein: `DigitalInput_CLK_I2`

- **Typ**: `logiBUS::io::DI::logiBUS_IE`
- **Parameter**:
  - `QI` = `TRUE`
  - `Input` = `Input_I2` (Hardware-Eingang I2)
  - `InputEvent` = `BUTTON_SINGLE_CLICK`
- **Ereignis-Ausgänge**: `IND`
- **Funktionsweise**: Wie oben, aber für den zweiten Taster an Eingang I2.

Programmablauf und Verbindungen

Die folgende Beschreibung erläutert den Signalfluss innerhalb der Subapplikation.

1. **Start/Stop des Taktgebers**
   - Ein Tastendruck an I1 löst ein Ereignis am Ausgang `IND` von `DigitalInput_CLK_I1` aus. Dieses Ereignis wird mit dem Ereignis-Eingang `START` des `E_CYCLE` verbunden → der Zyklusgenerator startet.
   - Ein Tastendruck an I2 löst ein Ereignis am Ausgang `IND` von `DigitalInput_CLK_I2` aus. Dieses Ereignis wird mit dem Ereignis-Eingang `STOP` des `E_CYCLE` verbunden → der Zyklusgenerator stoppt.

2. **Zyklus und Verteilung**
   - Der `E_CYCLE` erzeugt alle 10 ms ein Ereignis an seinem Ausgang `EO`.
   - Dieses Ereignis wird auf den Eingang `EI` des `E_SPLIT_4` gegeben. Der Splitter verteilt das Ereignis auf alle vier Ausgänge (`EO1` .. `EO4`).
3. **Zusammenführung**
   - Die vier Ausgänge des Splitters sind mit den vier Eingängen (`EI1` .. `EI4`) des `E_MERGE_4` verbunden. Dadurch wird jedes Ereignis, egal über welchen Pfad, sofort an den Ausgang `EO` des Mergers weitergeleitet.
   - Die Verbindung von Splitter zu Merger über alle vier Pfade dient hier als reine Durchleitung (Redundanz), könnte aber für spätere Erweiterungen genutzt werden.
4. **Toggle-Flipflop**
   - Das Ausgangsereignis des `E_MERGE_4` wird auf den Takteingang `CLK` des `E_T_FF` gegeben.
   - Bei jedem Takt wechselt der Ausgang `Q` des Flipflops seinen Zustand (0 ⟶ 1 ⟶ 0 ⟶ ...).
   - Gleichzeitig wird ein Ereignis am Ausgang `EO` des Flipflops ausgelöst.
5. **Ausgabe**
   - Das Ereignis `EO` des Flipflops wird mit dem Ereignis-Eingang `REQ` des Ausgangsbausteins `DigitalOutput_Q1` verbunden.
   - Der aktuelle Wert `Q` des Flipflops wird auf den Dateneingang `OUT` des Ausgangsbausteins gegeben.
   - Bei jedem Takt wird der Ausgang Q1 auf den aktuellen Flipflop-Zustand gesetzt – es entsteht ein Blinksignal mit einer Periodendauer von 20 ms (10 ms Ein, 10 ms Aus, wenn die Zykluszeit 10 ms beträgt).

Datenverbindungen

- `E_T_FF.Q` ⟶ `DigitalOutput_Q1.OUT` Überträgt den toggelnden Wert (0/1) auf den Ausgangsbaustein.

Ereignisverbindungen (Zusammenfassung)

Quelle	Ziel
`DigitalInput_CLK_I1.IND`	`E_CYCLE.START`
`DigitalInput_CLK_I2.IND`	`E_CYCLE.STOP`
`E_CYCLE.EO`	`E_SPLIT_4.EI`
`E_SPLIT_4.EO1`	`E_MERGE_4.EI1`
`E_SPLIT_4.EO2`	`E_MERGE_4.EI2`
`E_SPLIT_4.EO3`	`E_MERGE_4.EI3`
`E_SPLIT_4.EO4`	`E_MERGE_4.EI4`
`E_MERGE_4.EO`	`E_T_FF.CLK`
`E_T_FF.EO`	`DigitalOutput_Q1.REQ`

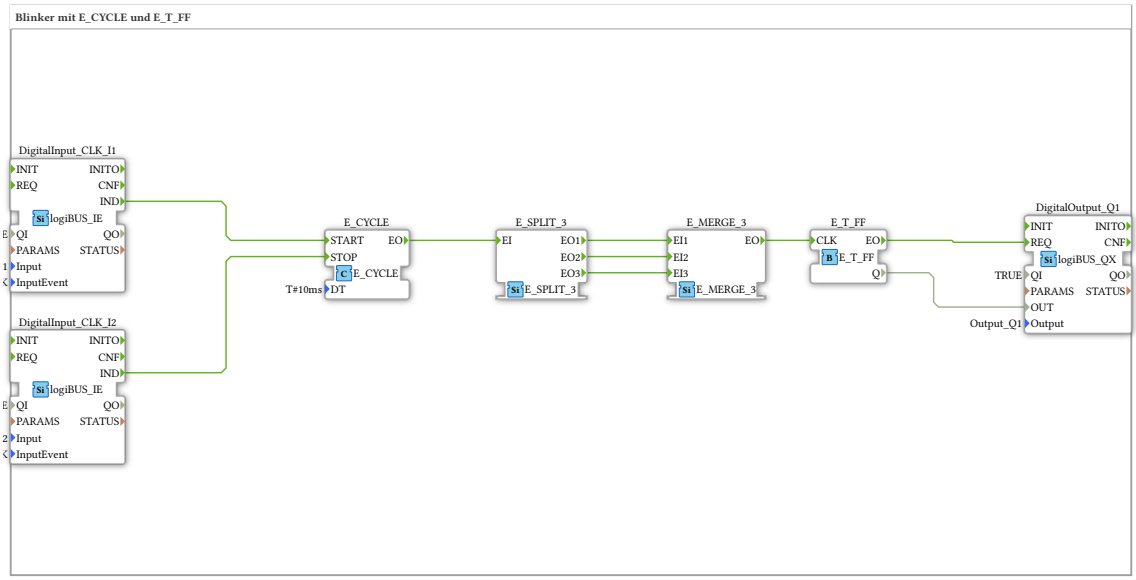
Zusammenfassung

Die Übung 007b demonstriert den Einsatz von zyklischer Ereigniserzeugung (`E_CYCLE`), Ereignisverteilung und -zusammenführung (`E_SPLIT_4`, `E_MERGE_4`) sowie einem Toggle-Flipflop (`E_T_FF`) zur Erzeugung eines Blinksignals. Die Steuerung erfolgt über zwei logiBUS-Taster (Start/Stop). Der Aufbau ist als wiederverwendbare Subapplikation realisiert und kann direkt in eine 4diac-IDE-Umgebung importiert werden. Die Schaltung ist ein grundlegendes Beispiel für zeitgesteuerte Ausgänge mit einfacher Benutzerinteraktion.

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

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

2.2.1.95 Uebung_007c: Blinker mit E_CYCLE und E_T_FF



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert die Erstellung eines einfachen Blinkers mithilfe der IEC-61499-Bausteine `E_CYCLE` und `E_T_FF`.
Der Blinker wird über zwei digitale Eingänge gesteuert:

- **Eingang I1** (Taster Single-Click) startet die Blinkfunktion.
- **Eingang I2** (Taster Single-Click) stoppt sie.

Der Ausgang **Q1** schaltet periodisch im 10-ms-Takt zwischen Ein und Aus um, solange die Blinkfunktion aktiv ist.
Die Übung vermittelt den Umgang mit zyklischen Ereignissen, Ereignis-Splitting/-Merging und der Toggle-Funktion.

Verwendete Funktionsbausteine (FBs)

Bausteinname	Typ	Parameter	Beschreibung
`E_CYCLE`	`iec61499::events::E_CYCLE`	`DT = T#10ms`	Erzeugt alle 10 ms ein Ereignis an seinem Ausgang `EO`.
`E_SPLIT_3`	`iec61499::events::E_SPLIT_3`	–	Verteilt ein eingehendes Ereignis auf drei identische Ausgänge (`EO1`, `EO2`, `EO3`).
`E_MERGE_3`	`iec61499::events::E_MERGE_3`	–	Fasst drei eingehende Ereignisse (`EI1`, `EI2`, `EI3`) zu einem gemeinsamen Ausgang `EO` zusammen.
`E_T_FF`	`iec61499::events::E_T_FF`	–	Toggle-Flipflop: Bei jedem Ereignis am Eingang `CLK` wechselt der Ausgang `Q` seinen Wert (0⟶1 oder 1⟶0).

`DigitalInput_CLK_I1`	`logiBUS::io::DI::logiBUS_IO`	`QI = TRUE, Input = Input_I1, InputEvent = BUTTON_SINGLE_CLICK`	Wandelt einen Tastendruck an I1 in ein Ereignis am Ausgang `IND` um (Start-Befehl).
`DigitalInput_CLK_I2`	`logiBUS::io::DI::logiBUS_IO`	`QI = TRUE, Input = Input_I2, InputEvent = BUTTON_SINGLE_CLICK`	Wandelt einen Tastendruck an I2 in ein Ereignis am Ausgang `IND` um (Stopp-Befehl).
`DigitalOutput_Q1`	`logiBUS::io::DQ::logiBUS_IO`	`QI = TRUE, Output = Output_Q1`	Steuert den digitalen Ausgang Q1. Ein Ereignis am Eingang `REQ` übernimmt den Datenwert am Eingang `OUT` und gibt ihn physikalisch aus.

Programmablauf und Verbindungen

1. **Start**: Ein Tastendruck an **I1** erzeugt ein Ereignis an `DigitalInput_CLK_I1.IND`. Dieses wird mit dem **START**-Eingang von `E_CYCLE` verbunden und aktiviert den zyklischen Timer.
2. **Stopp**: Ein Tastendruck an **I2** erzeugt ein Ereignis an `DigitalInput_CLK_I2.IND`. Dieses wird mit dem **STOP**-Eingang von `E_CYCLE` verbunden und deaktiviert den Timer.
3. **Zyklus**: Solange `E_CYCLE` aktiv ist, erzeugt es alle 10 ms ein Ereignis an seinem Ausgang `EO`.
4. **Splitting**: Dieses Ereignis wird durch `E_SPLIT_3` auf drei Ausgänge (`EO1`–`EO3`) verteilt.
5. **Merging**: Die drei identischen Ereignisse werden über `E_MERGE_3` wieder zu einem einzigen Ereignis zusammengeführt. (Dadurch wird die Verzögerungszeit minimal gehalten und die Signalintegrität sichergestellt.)
6. **Toggle**: Das gemergte Ereignis triggert den `CLK`-Eingang von `E_T_FF`. Der Zustand des Flipflops wechselt bei jedem Takt. Der aktuelle Wert wird am Datenausgang `Q` bereitgestellt.
7. **Ausgabe**: Das Ereignis `EO` von `E_T_FF` wird zum `REQ`-Eingang von `DigitalOutput_Q1` geführt. Gleichzeitig wird der Datenwert `Q` (0 oder 1) an den `OUT`-Eingang der Ausgangsbaugruppe übergeben. Bei jedem Takt wird der Ausgang Q1 entsprechend gesetzt.

Lernziele:

- Verständnis von zyklischen Ereignissen (`E_CYCLE`)
- Nutzung eines Toggle-Flipflops (`E_T_FF`)
- Ereignis-Vervielfachung und Zusammenführung (`E_SPLIT_3`, `E_MERGE_3`)
- Ein- und Ausschalten einer Funktion über digitale Eingänge

Schwierigkeitsgrad: Anfänger
Vorkenntnisse: Grundlagen der IEC-61499-Ereigniskette, einfache Verbindungen zwischen Funktionsbausteinen

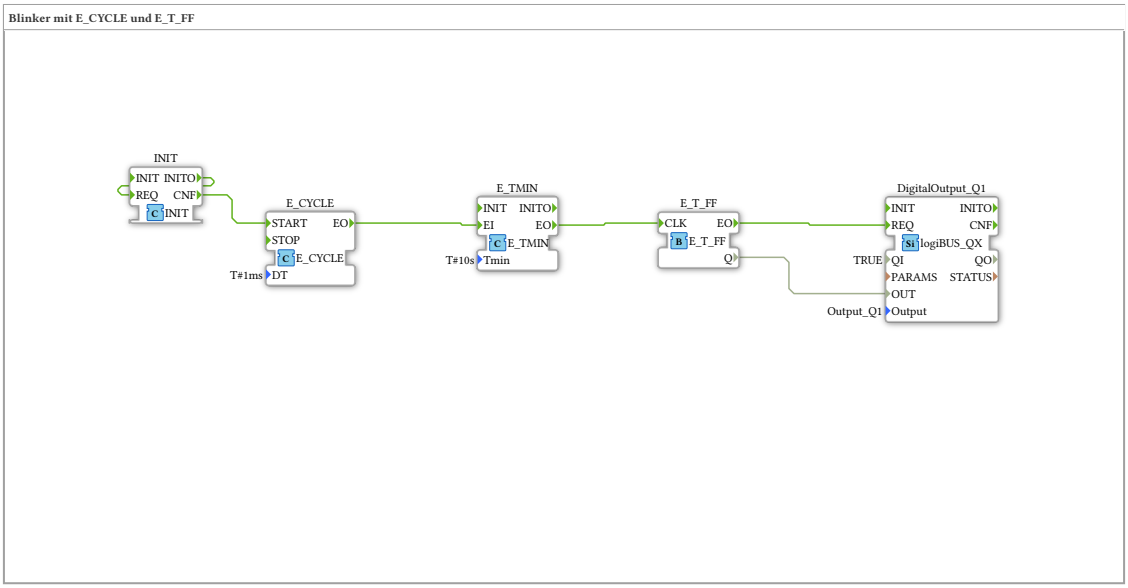
Hinweise zum Start:

- Die Übung verwendet die logiBUS-Hardware-Schnittstelle – stellen Sie sicher, dass die Eingänge I1 und I2 sowie der Ausgang Q1 korrekt angeschlossen sind.
- Die Taster müssen im **Single-Click**-Modus konfiguriert sein.
- Nach dem Aktivieren von E_CYCLE (Start) blinkt Q1 solange, bis der Stopp-Taster betätigt wird.

Zusammenfassung

Die Übung `Uebung_007c` realisiert einen an- und ausschaltbaren Blinker auf Basis von `E_CYCLE` und `E_T_FF`. Sie zeigt, wie systemzeitgesteuerte Ereignisse mit einem Toggle-Flipflop kombiniert werden, um einen wechselnden Ausgang zu erzeugen. Die Verwendung von `E_SPLIT_3` und `E_MERGE_3` verdeutlicht die Handhabung von Ereignisverzweigungen. Durch die beiden Taster wird der Anwender in die Lage versetzt, eine zyklische Funktion gezielt zu starten und zu stoppen.

2.2.1.96 Uebung_007d: Blinker mit E_CYCLE und E_T_FF



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung demonstriert die Realisierung eines Blinkers mithilfe der Funktionsbausteine `E_CYCLE` und `E_T_FF`. Der Blinker schaltet einen digitalen Ausgang in einem festen Zeitrhythmus ein und aus. Hierbei wird ein periodisches Ereignis durch `E_CYCLE` erzeugt, über `E_TMIN` auf eine bestimmte Mindestzeit verlängert und anschließend durch einen T-Flipflop (`E_T_FF`) getoggelt. Das Ergebnis wird auf einen Digitalausgang ausgegeben.

Verwendete Funktionsbausteine (FBs)

- **`E_CYCLE`** (Typ: `iec61499::events::E_CYCLE`) Erzeugt zyklisch ein Ereignis am Ausgang `EO`.
  - Parameter: `DT` = `T#1ms` (Zykluszeit 1 Millisekunde)
- **`E_TMIN`** (Typ: `iec61499::events::E_TMIN`) Gibt am Ausgang `EO` ein Ereignis aus, wenn seit dem letzten Eingangsereignis mindestens die Zeit `Tmin` vergangen ist.
  - Parameter: `Tmin` = `T#10s`
- **`E_T_FF`** (Typ: `iec61499::events::E_T_FF`) Ein T-Flipflop mit Takteingang `CLK`. Bei jedem Ereignis an `CLK` wechselt der Datenausgang `Q` seinen Zustand (0⟶1, 1⟶0). Der Ereignisausgang `EO` wird gleichzeitig mit dem `CLK`-Ereignis aktiviert.
- **`DigitalOutput_Q1`** (Typ: `logiBUS::io::DQ::logiBUS_QX`) Digitaler Ausgangsbaustein, der den Wert des Eingangs `OUT` an den physischen Ausgang `Output_Q1` weitergibt.
  - Parameter: `QI` = `TRUE` (Aktivierung), `Output` = `Output_Q1` (Hardware-Adresse)
  - Ereigniseingang `REQ`: Übernahme des Datenwertes

Programmablauf und Verbindungen

Der Ablauf der Übung wird durch die Ereignis- und Datenverbindungen im Netzwerk bestimmt:

1. **Ereignis-Erzeugung**: `E_CYCLE` sendet alle 1 ms ein Ereignis an den Eingang `EI` von `E_TMIN`.
2. **Zeitverzögerung**: `E_TMIN` wartet, bis seit seinem letzten Ereignis mindestens 10 Sekunden vergangen sind. Erst dann gibt es selbst ein Ereignis an seinem Ausgang `EO` aus. Dadurch wird das ursprüngliche 1-ms-Ereignis auf eine Periode von 10 Sekunden „verlangsamt“.
3. **Toggle-Flipflop**: Das verzögerte Ereignis von `E_TMIN.EO` wird an den Takteingang `CLK` von `E_T_FF` geführt. Jeder Takt wechselt den Datenausgang `Q` des Flipflops. Gleichzeitig erzeugt `E_T_FF` ein Ereignis an `EO`.
4. **Ausgabe**:

- Der Datenwert `E_T_FF.Q` wird über eine Datenverbindung an den Dateneingang `OUT` von `DigitalOutput_Q1` weitergeleitet.
- Das Ereignis `E_T_FF.EO` triggert den `REQ`-Eingang des Digitalausgangs, sodass der aktuelle Wert von `OUT` übernommen und am physischen Ausgang gesetzt wird.

Ergebnis: Der digitale Ausgang schaltet im 10-Sekunden-Takt: 10 Sekunden ein, 10 Sekunden aus (Blinkfrequenz 0,05 Hz).

Diese Übung vermittelt Grundlagen der ereignisgesteuerten Zeitsteuerung in 4diac. Sie zeigt, wie aus einem schnellen Zyklus durch `E_TMIN` ein langsamerer Takt erzeugt und mit einem T-Flipflop in ein Rechtecksignal umgewandelt werden kann.

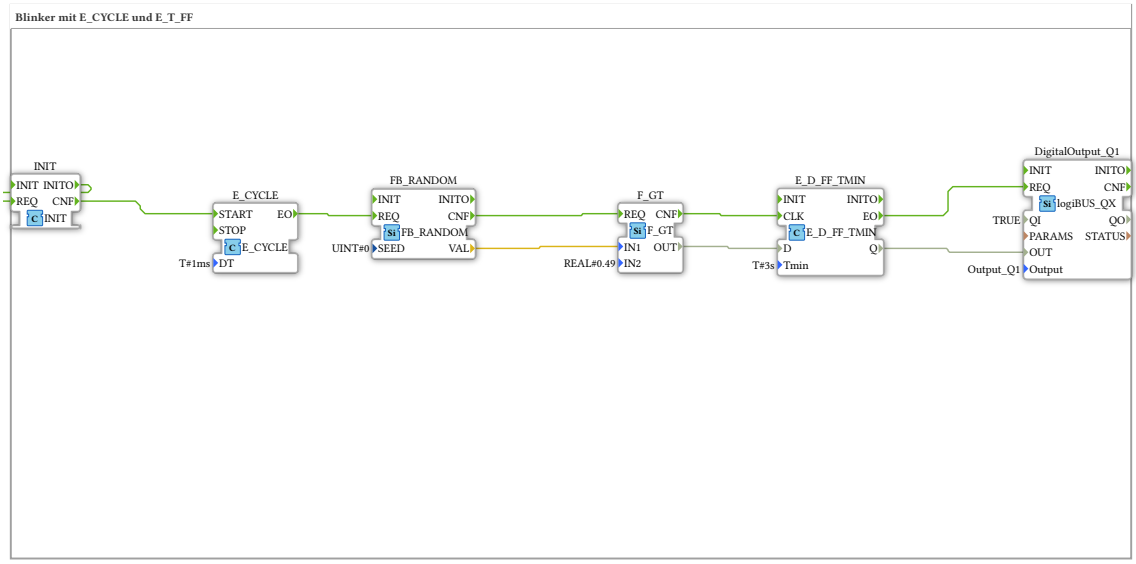
Zusammenfassung

Die Übung **Uebung_007d** realisiert einen einfachen Blinker unter Verwendung der Bausteine `E_CYCLE`, `E_TMIN` und `E_T_FF`. Die Kaskadierung von Ereignisbausteinen ermöglicht eine flexible Zeitsteuerung, ohne auf herkömmliche Timer oder Zähler angewiesen zu sein. Der erstellte Subapplikationstyp kann direkt in übergeordnete Projekte eingebunden werden, um z. B. Signallampen oder optische Anzeigen zeitgesteuert zu betreiben.

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

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

2.2.1.97 Uebung_007d2: Blinker mit E_CYCLE und E_T_FF



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung realisiert einen Blinker unter Verwendung von `E_CYCLE` und `E_D_FF_TMIN`. Der Blinker erzeugt ein zufälliges Ein-/Ausschaltverhältnis, wobei ein Flipflop mit minimaler Einschaltzeit von 3 Sekunden sicherstellt, dass der Ausgang nach einem Einschalten nicht zu früh wieder ausschaltet.

Verwendete Funktionsbausteine (FBs)

- **E_CYCLE** *Typ*: `iec61499::events::E_CYCLE` *Parameter*: `DT = T#1ms` (Zykluszeit 1 ms) Ein zyklischer Ereignisgenerator, der periodisch ein Ereignis auslöst.
- **FB_RANDOM** *Typ*: `eclipse4diac::utils::FB_RANDOM` *Parameter*: `SEED = 0` (Startwert für Zufallszahl) Erzeugt einen Zufallswert im Bereich 0..1.
- **F_GT** *Typ*: `iec61131::comparison::F_GT` *Parameter*: `IN2 = REAL#0.49` (fester Vergleichswert) Vergleicht die Werte `IN1 > IN2` und liefert einen booleschen Ausgang.

- **E_D_FF_TMIN** *Typ*: `iec61499::events::E_D_FF_TMIN` *Parameter*: `Tmin = T#3s` (minimale Einschaltzeit) Ein D-Flipflop, das nach einem steigenden Takt den Dateneingang übernimmt und den Ausgang für mindestens `Tmin` auf TRUE hält.
- **DigitalOutput_Q1** *Typ*: `logiBUS::io::DQ::logiBUS_QX` *Parameter*: `QI = TRUE` (Aktivierung des Ausgangs), `Output = Output_Q1` (physikalischer Ausgang) Digitaler Ausgang auf dem logiBUS-System.

Programmablauf und Verbindungen

Der Ablauf wird durch den zyklischen Ereignisgenerator `E_CYCLE` (alle 1 ms) gestartet. Das Ereignis `EO` triggert den Baustein `FB_RANDOM` (`REQ`), der einen zufälligen Gleitkommawert (`VAL`) erzeugt. Dieser Wert wird an den DatenEingang `IN1` von `F_GT` übergeben. Der Vergleichsbaustein vergleicht den Zufallswert mit dem festen Schwellwert `0.49` und erzeugt ein boolesches Ergebnis (`OUT`). Das Ergebnis `TRUE` entspricht einer Zufallszahl > 0.49, `FALSE` entspricht ≤ 0.49.

Das Ereignis `CNF` von `FB_RANDOM` triggert `F_GT` (`REQ`). Nach der Berechnung sendet `F_GT` ein `CNF` an den Takteingang `CLK` von `E_D_FF_TMIN`. Das Flipflop übernimmt gleichzeitig den aktuellen Wert von `D` (entspricht `F_GT.OUT`). Der Ausgang `Q` des Flipflops bleibt nach einem Einschalten mindestens 3 s lang auf `TRUE`, selbst wenn `D` vor Ablauf dieser Zeit wieder auf `FALSE` wechselt. Der Ausgang `Q` steuert den Dateneingang `OUT` von `DigitalOutput_Q1`. Das Ereignis `EO` des Flipflops triggert den Ausgangsbaustein (`REQ`), sodass der physikalische Ausgang den aktuellen Wert anlegt.

Ereignis- und Datenverbindungen:

- `E_CYCLE.EO` ⟶ `FB_RANDOM.REQ`
- `FB_RANDOM.CNF` ⟶ `F_GT.REQ`
- `FB_RANDOM.VAL` ⟶ `F_GT.IN1`
- `F_GT.CNF` ⟶ `E_D_FF_TMIN.CLK`
- `F_GT.OUT` ⟶ `E_D_FF_TMIN.D`
- `E_D_FF_TMIN.EO` ⟶ `DigitalOutput_Q1.REQ`
- `E_D_FF_TMIN.Q` ⟶ `DigitalOutput_Q1.OUT`

Zusammenfassung

Die Übung demonstriert die Kombination eines zyklischen Ereignisgenerators, eines Zufallszahlengenerators, eines Vergleichsbausteins, eines Flipflops mit minimaler Einschaltzeit und eines digitalen Ausgangs. Lernziele sind:

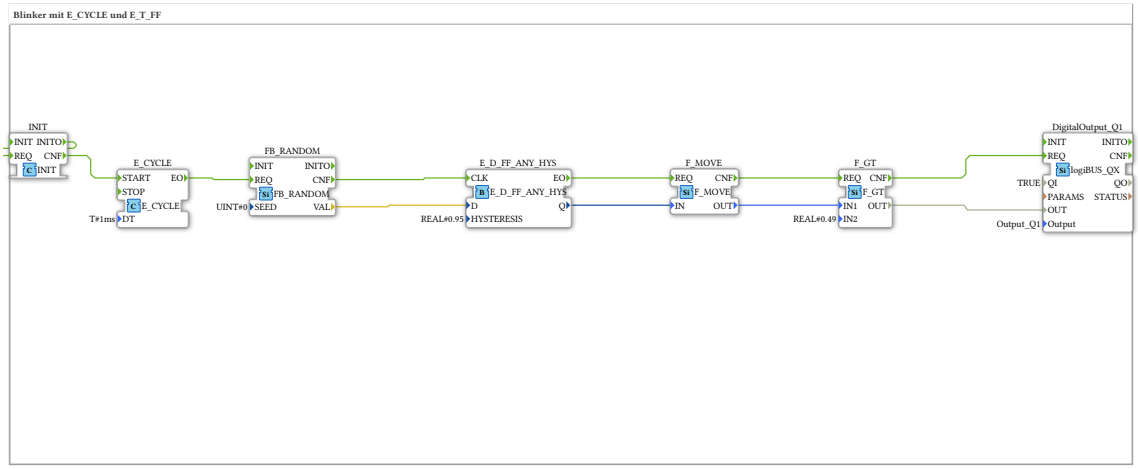
- Verständnis von Ereignis- und Datenflüssen in IEC 61499.
- Parametrierung von Zeitbausteinen (`E_CYCLE`, `E_D_FF_TMIN`).
- Erstellung eines Blinkmusters mit variablem Tastverhältnis durch einen Zufallsprozess.
- Integration eines logiBUS-Ausgangs.

Vorausgesetzt werden Grundkenntnisse der Ereignisbehandlung nach IEC 61499 und der logiBUS-Ausgangsansteuerung.

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

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

2.2.1.98 Uebung_007d3: Blinker mit E_CYCLE und E_T_FF



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung realisiert einen zufallsgesteuerten Blinker mit Hilfe der Funktionsbausteine `E_CYCLE`, `FB_RANDOM`, `E_D_FF_ANY_HYS`, `F_MOVE` und `F_GT`. Ein zyklischer Takt triggert die Erzeugung eines Zufallswerts, der über ein Flip-Flop mit Hysterese und einen Komparator einen digitalen Ausgang schaltet. Der Blinker simuliert damit ein unregelmäßiges Ein-/Ausschaltverhalten.

Verwendete Funktionsbausteine (FBs)

Bausteinname	Typ	Parameter	Kurzbeschreibung
`E_CYCLE`	`iec61499::events::E_CYCLE`	DT = T#1ms	Erzeugt alle 1 ms ein Ereignis am Ausgang `EO`.
`FB_RANDOM`	`eclipse4diac::utils::FB_RANDOM`	SEED = 0	Liefert bei jedem `REQ`-Ereignis einen neuen REAL-Zufallswert zwischen 0 und 1 am Ausgang `VAL`.
`E_D_FF_ANY_HYS`	`iec61499::signalprocessing::hysteresis::E_D_FF_ANY_HYS`	HYSTERESIS = REAL#0.95	Getaktetes Flip-Flop mit Hysterese: Der Eingang `D` wird mit dem Ereignis an `CLK` übernommen. Der Ausgang `Q` schaltet erst um, wenn der Wert die Hysterese überschreitet.
`F_MOVE`	`iec61131::selection::F_MOVE`	DataType = REAL	Kopiert den Eingangswert (`IN`) unverändert auf den Ausgang (`OUT`). Dient hier der Typwandlung von BOOL → REAL.
`F_GT`	`iec61131::comparison::F_GT`	IN2 = REAL#0.49	Vergleicht `IN1` mit der Konstanten `IN2` und gibt am Ausgang `OUT` TRUE aus, wenn `IN1` größer als 0.49 ist.

DigitalOutput_Q1	logiBUS::io::DQ::logiBUS_QX0.1	QI = TRUE, Output = Output_Q1	Digitalausgangsbaustein, der das übergebene Signal (OUT) an die physikalische Adresse Output_Q1 weitergibt.

Programmablauf und Verbindungen

1. **Taktgenerierung** E_CYCLE erzeugt alle 1 ms ein Ereignis an seinem Ausgang EO.
2. **Zufallswert erzeugen** Dieses Ereignis wird über die Eventverbindung an den Eingang REQ von FB_RANDOM weitergegeben. FB_RANDOM berechnet einen zufälligen REAL-Wert zwischen 0 und 1 und gibt ihn am Datenausgang VAL aus. Gleichzeitig signalisiert es die Fertigstellung über CNF.
3. **Flip-Flop mit Hysterese** Das CNF-Ereignis triggert den Takteingang CLK von E_D_FF_ANY_HYS. Der Datenwert von FB_RANDOM.VAL wird an den Dateneingang D gelegt. Durch die eingestellte Hysterese von 0.95 wird der Ausgang Q nur dann auf TRUE gesetzt, wenn der Zufallswert die vorherige Schwelle deutlich überschreitet; bei Unterschreitung erfolgt das Zurücksetzen mit entsprechender Verzögerung. Der Ausgang Q ist ein BOOL-Wert.
4. **Typwandlung** Nach dem Flip-Flop wird der BOOL-Wert über F_MOVE (mit DataType = REAL) in eine REAL-Zahl umgewandelt (TRUE ⟶ 1.0, FALSE ⟶ 0.0). Das Ereignis dafür liefert E_D_FF_ANY_HYS.EO.
5. **Vergleich mit Schwellwert** Der umgewandelte Wert gelangt über die Datenverbindung an den Eingang IN1 von F_GT. Dieser vergleicht ihn mit der Konstanten IN2 = 0.49. Ist IN1 größer, wird der Ausgang OUT auf TRUE gesetzt, andernfalls auf FALSE.
6. **Ausgabe an Digitalausgang** Das Ergebnis des Vergleichs (F_GT.OUT) wird sowohl als Datenwert an den Eingang OUT des Digitalausgangsbausteins DigitalOutput_Q1 gelegt, als auch über die Eventverbindung (F_GT.CNF → DigitalOutput_Q1.REQ) die Verarbeitung angestoßen. DigitalOutput_Q1 gibt den Wert schließlich an die physikalische Leitung Output_Q1 aus.

Der gesamte Ablauf wiederholt sich mit jedem Takt von E_CYCLE (alle 1 ms), sodass am Ausgang ein unregelmäßiges Blinksignal entsteht, das durch die Hysterese verzögert und durch den Schwellwert weiter geformt wird.

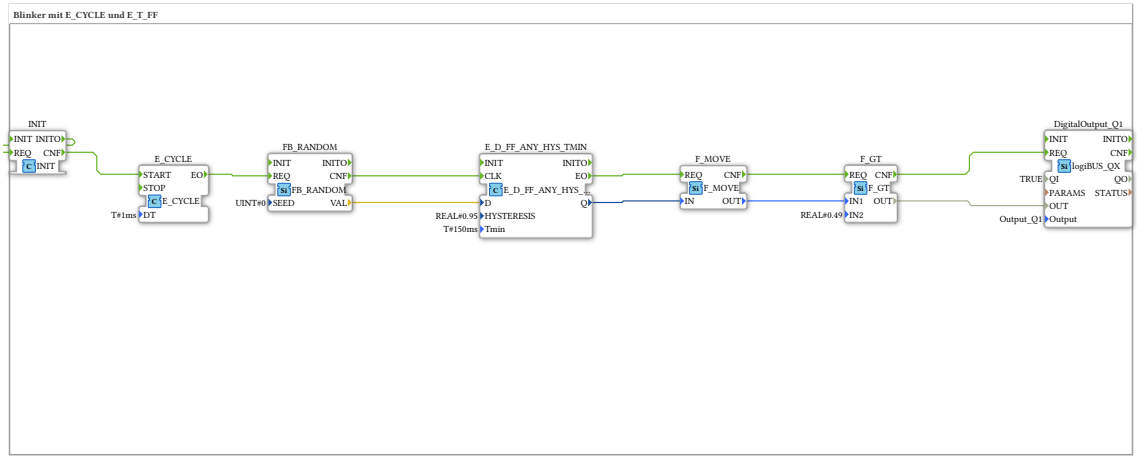
Zusammenfassung

Die Übung demonstriert die Kombination von zyklischer Ereignissteuerung, Zufallswertgenerierung, Flip-Flop mit Hysterese, Typkonvertierung und Vergleichslogik zur Erzeugung eines dynamischen Ausgangssignals. Lernziele sind das Verständnis von Ereignis-Daten-Verbindungen, die Parametrierung von Zeit- und Hysterese-Bausteinen sowie die Zusammenschaltung mehrerer Funktionsbausteine zu einer funktionalen Einheit. Voraussetzung sind Grundkenntnisse in IEC 61499 und der 4diac-IDE.

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

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

2.2.1.99 Uebung_007d4: Blinker mit E_CYCLE und E_T_FF



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung realisiert einen **Blinker** basierend auf einem zufälligen Signal. Ein periodischer Takt (E_CYCLE) triggert einen Zufallsgenerator (FB_RANDOM). Dessen Ausgang wird durch einen Hysterese-Flipflop (E_D_FF_ANY_HYS) gefiltert, anschließend über einen Move-Baustein aufbereitet und mit einem Schwellwert verglichen (F_GT). Das Ergebnis schaltet einen digitalen Ausgang (logiBUS QX) – es entsteht ein unregelmäßig blinkendes Signal.

Verwendete Funktionsbausteine (FBs)

- **DigitalOutput_Q1** (Typ: `logiBUS::io::DQ::logiBUS_QX`)
  - Parameter: `QI` = TRUE, `Output` = „Output_Q1“
  - Zweck: physischer Digitalausgang (hier simuliert).
- **E_CYCLE** (Typ: `iec61499::events::E_CYCLE`)
  - Parameter: `DT` = T#1ms
  - Zweck: erzeugt einen periodischen Ereignisimpuls (Takt).
- **FB_RANDOM** (Typ: `eclipse4diac::utils::FB_RANDOM`)
  - Parameter: `SEED` = 0
  - Zweck: erzeugt bei jedem REQ einen neuen Zufallswert (REAL) am Ausgang VAL.
- **F_GT** (Typ: `iec61131::comparison::F_GT`)
  - Parameter: `IN2` = REAL#0.49
  - Zweck: vergleicht zwei REAL-Werte; OUT ist TRUE wenn IN1 > IN2.
- **E_D_FF_ANY_HYS** (Typ: `logiBUS::signalprocessing::hysteresis::E_D_FF_ANY_HYS_TMIN`)
  - Parameter: `HYSTERESIS` = REAL#0.95, `Tmin` = T#150ms
  - Zweck: hysteresebehafteter D-Flipflop mit Mindesteinschaltzeit; Q wird TRUE wenn D > Hysterese und der letzte Zustand mindestens Tmin zurückliegt.
- **F_MOVE** (Typ: `iec61131::selection::F_MOVE`)
  - Attribut: `DataType` = REAL
  - Zweck: kopiert den Wert von IN nach OUT (ohne Verzögerung).

Programmablauf und Verbindungen

1. **Taktgenerierung** E_CYCLE erzeugt alle 1 ms ein Ereignis an seinem Ausgang `EO`. Dieses Ereignis wird direkt an den Eingang `REQ` von FB_RANDOM weitergeleitet.
2. **Zufallswert erzeugen** FB_RANDOM liefert bei jedem `REQ` einen neuen REAL-Zufallswert an `VAL`. Dieser Wert wird dem Dateneingang `D` des Hysterese-Flipflops zugeführt.
3. **Hysterese-Filter** E_D_FF_ANY_HYS vergleicht den aktuellen Wert mit der Hysterese-Schwelle (0.95). Überschreitet der Wert die Schwelle **und** ist die Mindestzeit Tmin seit dem letzten Umschalten vergangen, wird der Ausgang `Q` auf TRUE gesetzt, ansonsten auf FALSE. Ein gültiger Wechsel erzeugt ein Ereignis an `EO`.
4. **Signalweitergabe**

- Das Ereignis `EO` vom Hysterese-Flipflop triggert den F_MOVE-Baustein.
- F_MOVE kopiert den aktuellen Zustand `Q` (als REAL: 0.0 oder 1.0?) an seinen Ausgang `OUT`.
- Der Ausgang `OUT` wird an den Dateneingang `IN1` des Vergleichsbausteins F_GT übergeben.
- Das Ereignis `CNF` von F_MOVE startet F_GT.

5. **Schwellwertvergleich** F_GT prüft, ob der kopierte Wert größer als 0.49 ist.
   - Ist der Wert > 0.49 (d.h. der Hysterese-Ausgang war TRUE), wird `OUT` = TRUE.
   - Ist er ≤ 0.49, wird `OUT` = FALSE. Das Ereignis `CNF` von F_GT triggert den Digitalausgang.
6. **Ausgang setzen** Der Digitalausgang `DigitalOutput_Q1` erhält über den Dateneingang `OUT` den Vergleichsergebnis. Der Ausgang des Bausteins (z.B. eine LED) leuchtet, wenn der aktuelle Zufallswert die Hysterese überschritten hat und der Vergleich positiv war.

Zusammenfassung der Signalverarbeitung:
`E_CYCLE` ⟶ `FB_RANDOM` ⟶ `E_D_FF_ANY_HYS` ⟶ `F_MOVE` ⟶ `F_GT` ⟶ `DigitalOutput_Q1`
Lernziele dieser Übung:

- Verständnis des Zusammenspiels von Ereignis- und Datenflüssen in 4diac.
- Einsatz eines zyklischen Taktes (E_CYCLE) zur Steuerung einer wiederholten Berechnung.
- Anwendung von Zufallsgeneratoren und Hysterese-Funktionen zur Erzeugung eines unregelmäßigen Signals.
- Praktische Nutzung von Vergleichsbausteinen und Ausgangsbausteinen.

Zusammenfassung

Die Übung „Uebung_007d4“ demonstriert den Aufbau eines **Blinkers mit zufälligen Einschaltphasen**. Ein periodischer Takt löst eine Kette von Bausteinen aus: Zufallsgenerator, Hysterese-Filter, Wertweitergabe, Schwellwertvergleich und schließlich einen Digitalausgang. Durch die Kombination von Hysterese und Mindestzeit entsteht ein Blinkmuster, das nicht rein zufällig ist, sondern gewisse Ein- und Ausschaltmindestzeiten einhält. Dieses Beispiel vertieft das Verständnis von ereignisgesteuerten Funktionsbausteinen, Parameterkonfiguration und der Kopplung von Daten- und Ereignisverbindungen in der 4diac-IDE.

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

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

2.2.1.100 Uebung_008: Blinker mit E_CYCLE, E_SWITCH und E_SR



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_008`. Hier wird die Logik eines dauerhaft laufenden Taktgebers mit internem Speicherzustand gezeigt.

Ziel der Übung

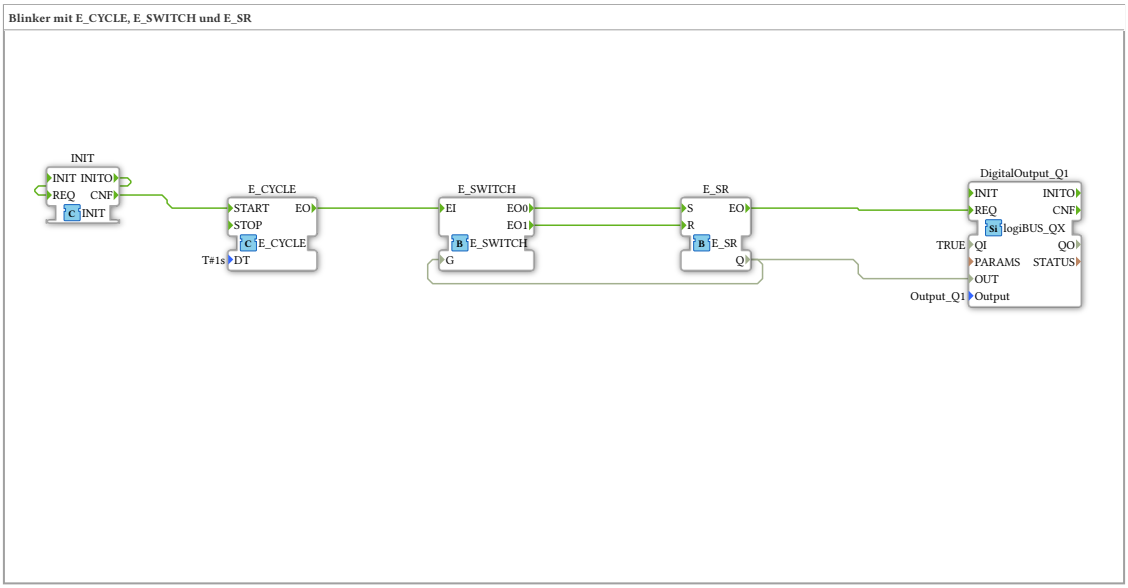
Implementierung eines autarken Blink-Schaltkreises.

Beschreibung und Komponenten

Die Subapplikation `Uebung_008.SUB` nutzt die Kombination aus `E_CYCLE`, `E_SWITCH` und `E_SR` ohne externe Steuereingänge.

Der Taktgeber `E_CYCLE` läuft (nach einmaliger Initialisierung durch das System) permanent durch. Die Logik sorgt dafür, dass der Ausgang `Q1` im Sekundentakt zwischen `TRUE` und `FALSE` wechselt. Da keine Stopp-Logik vorhanden ist, dient dieser Aufbau als permanenter Herzschlag des Programms.

Anwendungsbeispiel



Status-LED (Heartbeat): Eine LED direkt auf der CPU-Platine, die ständig blinkt, solange die Versorgungsspannung anliegt und das Steuerungsprogramm („Task“) fehlerfrei abgearbeitet wird. Hört die LED auf zu blinken, weiß der Techniker sofort, dass die Steuerung abgestürzt ist oder im Stopp-Zustand verharrt.

2.2.1.101 Uebung_009: Ticker

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_009`. Hier verbinden wir die Zeitbasis mit einer Zählfunktion und einer numerischen Anzeige.

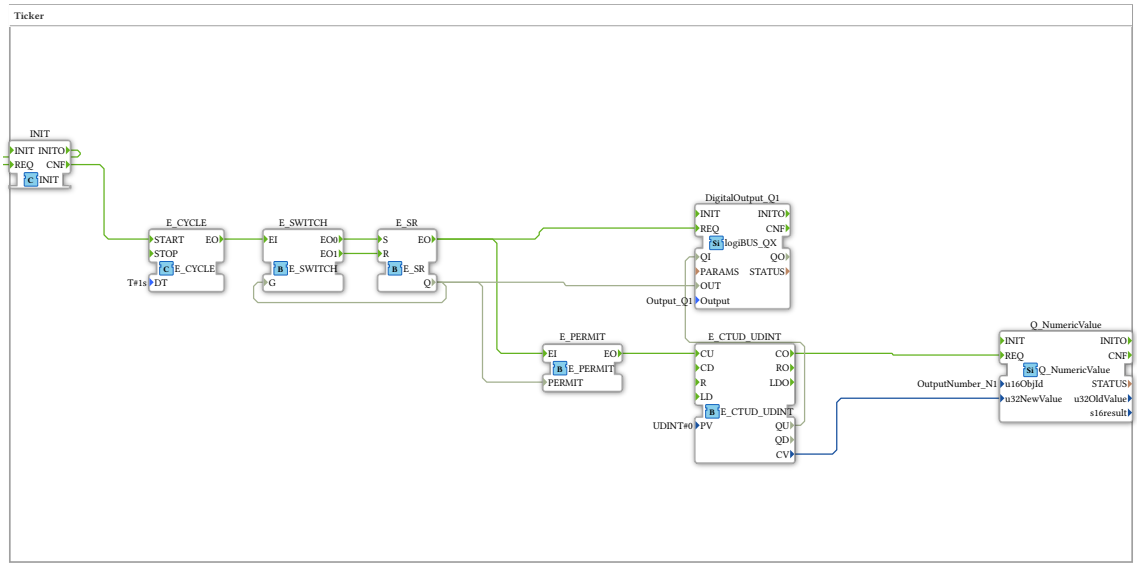
Ziel der Übung

Erlernen der ereignisbasierten Zählung (`E_CTUD`) und der Darstellung von Werten auf einem Terminal.

Beschreibung und Komponenten

In `Uebung_009.SUB` wird ein Taktgeber genutzt, um einen Aufwärtszähler anzusteuern, dessen Wert an ein ISOBUS-Terminal gesendet wird.

Funktionsbausteine (FBs)



- **`E_CYCLE` & `E_SR`**: Erzeugen einen permanenten Takt (wie in Übung 008).
- **`E_PERMIT`**: Ein Ereignis-Gatter. Es lässt Ereignisse am Eingang `EI` nur dann zum Ausgang `EO` durch, wenn der Dateneingang `PERMIT` auf `TRUE` steht.
- **`E_CTUD_UDINT`**: Ein Vorwärts-/Rückwärtszähler für große Ganzzahlen.
- **`Q_NumericValue`**: Ein ISOBUS-Ausgangsbaustein zur Anzeige einer Zahl auf dem Bildschirm.

Funktionsweise

1. Der Blinker-Teil erzeugt jede Sekunde ein Ereignis.
2. Dieses Ereignis wird durch `E_PERMIT` gefiltert. Da `PERMIT` mit dem blinkenden Ausgang verbunden ist, wird nur **jedes zweite** Ereignis (nämlich nur, wenn der Blinker gerade AN ist) durchgelassen.
3. Die durchgelassenen Events erreichen den Eingang `CU` (Count Up) des Zählers.
4. Der Zählerstand erhöht sich.
5. Bei jeder Änderung (`CO` - Count Output) wird der neue Wert an `Q_NumericValue` gesendet.
6. Auf dem ISOBUS-Terminal sieht der Nutzer eine Zahl, die stetig ansteigt.

Anwendungsbeispiel

Betriebsstundenzähler: Die Steuerung zählt die Zeitintervalle, in denen eine bestimmte Bedingung (z.B. „Motor läuft“) erfüllt ist. Der summierte Wert wird dauerhaft gespeichert und dem Bediener als Wartungsinformation am Terminal angezeigt.

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

- 🌐 E_CTU Event Counter Baustein auf ms-muc-docs.de

2.2.1.102 Uebung_009a: RampLimitFS

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_009a`. Hier wird die komplexe Steuerung eines Zahlenwertes über verschiedene Taster-Interaktionen demonstriert.

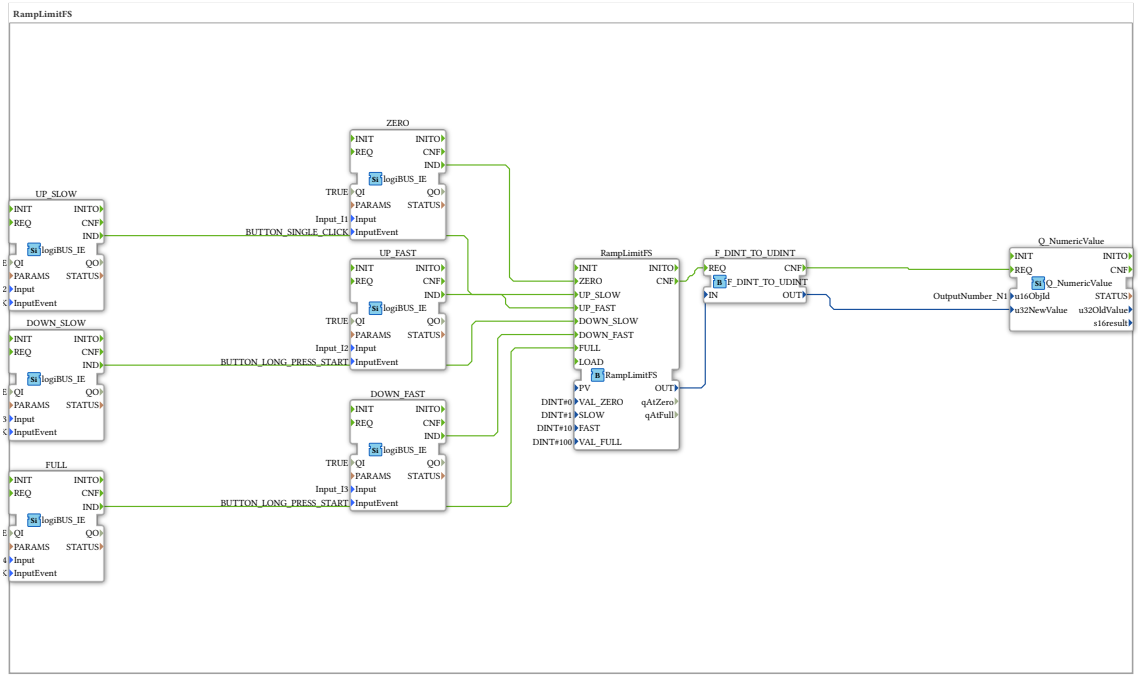
Ziel der Übung

Ansteuerung eines Rampen-Bausteins (`RampLimitFS`). Es wird gezeigt, wie verschiedene Ereignistypen (Klick vs. Langer Druck) genutzt werden können, um die Geschwindigkeit einer Wertänderung zu beeinflussen.

Beschreibung und Komponenten

Die Subapplikation `Uebung_009a.SUB` nutzt einen Rampen-Baustein zur stufenlosen Steuerung eines numerischen Werts zwischen 0 und 100.

Funktionsbausteine (FBs)



- **`RampLimitFS`**: Der Hauptbaustein aus der Signalverarbeitungs-Bibliothek. Er berechnet einen Ausgangswert, der sich zeitlich gleitend (Rampe) verändert.
- **Eingangstaster**:
  - `ZERO`: Setzt den Wert sofort auf 0.
  - `FULL`: Setzt den Wert sofort auf 100.
  - `UP_SLOW` (Klick): Erhöht den Wert langsam.
  - `UP_FAST` (Langer Druck): Erhöht den Wert schnell.
  - `DOWN_SLOW` (Klick): Verringert den Wert langsam.
  - `DOWN_FAST` (Langer Druck): Verringert den Wert schnell.

Funktionsweise

Der Rampen-Baustein reagiert auf unterschiedliche Event-Eingänge:

1. **Statische Ziele**: Bei `ZERO` oder `FULL` springt die interne Berechnung sofort auf die Grenzwerte.
2. **Dynamische Änderung**:
   - Ein Klick (`SINGLE_CLICK`) an `I2` triggert den `UP_SLOW` Eingang des Rampen-Bausteins. Der Wert steigt mit der im Parameter `SLOW` hinterlegten Rate.
   - Hält der Nutzer den Taster länger gedrückt (`LONG_PRESS_START`), wird der Eingang `UP_FAST` getriggert. Der Wert steigt nun wesentlich schneller (Parameter `FAST`).

Das Ergebnis wird am ISOBUS-Terminal als Zahl (`OutputNumber_N1`) angezeigt.

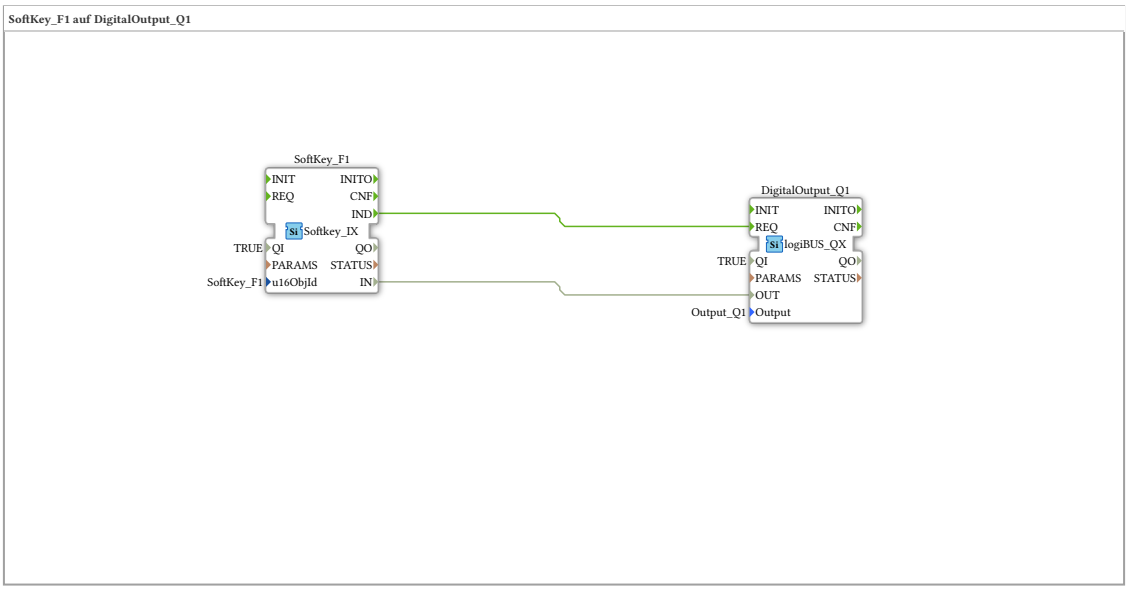
Anwendungsbeispiel

Elektrische Geschwindigkeits-Verstellung (Tempomat): Mit kurzen Tastendrücken am Joystick kann der Fahrer die Zielgeschwindigkeit in 1-km/h-Schritten feinjustieren. Hält er die Taste gedrückt, beschleunigt das Fahrzeug zügig bis zur Maximalgeschwindigkeit. Ein Tastendruck auf „Null“ bremst die Maschine sofort ab. Die Rampe sorgt dabei für weiche Übergänge und schont die Mechanik.

2.2.1.103 Uebung_010: SoftKey_F1 auf DigitalOutput_Q1

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_010`. Hier wird die Anbindung virtueller Bedienelemente eines ISOBUS-Terminals (Universal Terminal, UT) an physische Ausgänge demonstriert.

Podcast



- Das Kettenmonster erwacht: Lanz Bulldog Raupe – Die faszinierende Wiederbelebung des 10-Liter-Glühkopf-Arbeitstiers nach 25 Jahren Stillstand
- JBC Lötspitzen C470 vs. C245 vs. C210 vs. C115: Welche Spitze ist der Allrounder und wann brauchst du den Nano-Spezialisten?
- KI-Agenten revolutionieren Embedded-Entwicklung in 10 Stufen
- Miniware TS101: Das mobile Löt-Multitalent – Stärken, Schwächen und die USB-C Revolution
- Zwei WLANs gleichzeitig in Windows 10: Die geniale USB-Stick-Lösung für IoT-Geräte ohne Internet-Unterbrechung

---

Ziel der Übung

Verwendung eines `Softkey`-Bausteins zur direkten Steuerung eines digitalen Ausgangs. Es wird gezeigt, wie Ereignis- und Datenverbindungen genutzt werden, um eine Interaktion am Touchscreen in eine physische Aktion umzusetzen.

Beschreibung und Komponenten

Die Subapplikation `Uebung_010.SUB` verbindet eine Softkey-Instanz mit einem Standard-Ausgangsbaustein.

Funktionsbausteine (FBs)

- **`SoftKey_F1`**: Typ `isobus::UT::io::Softkey::Softkey_IX`. Dieser Baustein repräsentiert eine der Tasten am Bildschirmrand oder auf dem Touch-Display des ISOBUS-Terminals.
- **`DigitalOutput_Q1`**: Der physische Ausgang (z.B. ein Relais oder eine Lampe).

Parameter

- **`u16ObjId`**: Diese Kennung verweist auf das entsprechende Objekt im ISOBUS-Pool (hier `SoftKey_F1`).

Funktionsweise

Die Kommunikation erfolgt über die standardmäßige Trennung von Trigger und Wert:

```
<EventConnections>
    <Connection Source="SoftKey_F1.IND" Destination="DigitalOutput_Q1.REQ"/>
</EventConnections>
<DataConnections>
    <Connection Source="SoftKey_F1.IN" Destination="DigitalOutput_Q1.OUT"/>
</DataConnections>
```

Wenn der Bediener den Softkey am Terminal drückt:

1. Der Baustein `SoftKey_F1` erkennt die Betätigung über das CAN-Netzwerk.
2. Er setzt den Datenausgang `IN` auf `TRUE` und feuert ein `IND`-Event.
3. `DigitalOutput_Q1` empfängt den Trigger und schaltet den Hardware-Ausgang ein.
4. Beim Loslassen wechselt der Zustand zurück auf `FALSE`, ein erneutes Event wird gesendet und der Ausgang schaltet ab.

Anwendungsbeispiel

Hydraulikventil manuell steuern: Der Fahrer wählt auf seinem Terminal eine Service-Seite aus. Dort befindet sich ein Button „Ventil spülen“. Solange er diesen Button gedrückt hält, wird das entsprechende Magnetventil (`Q1`) angesteuert.

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

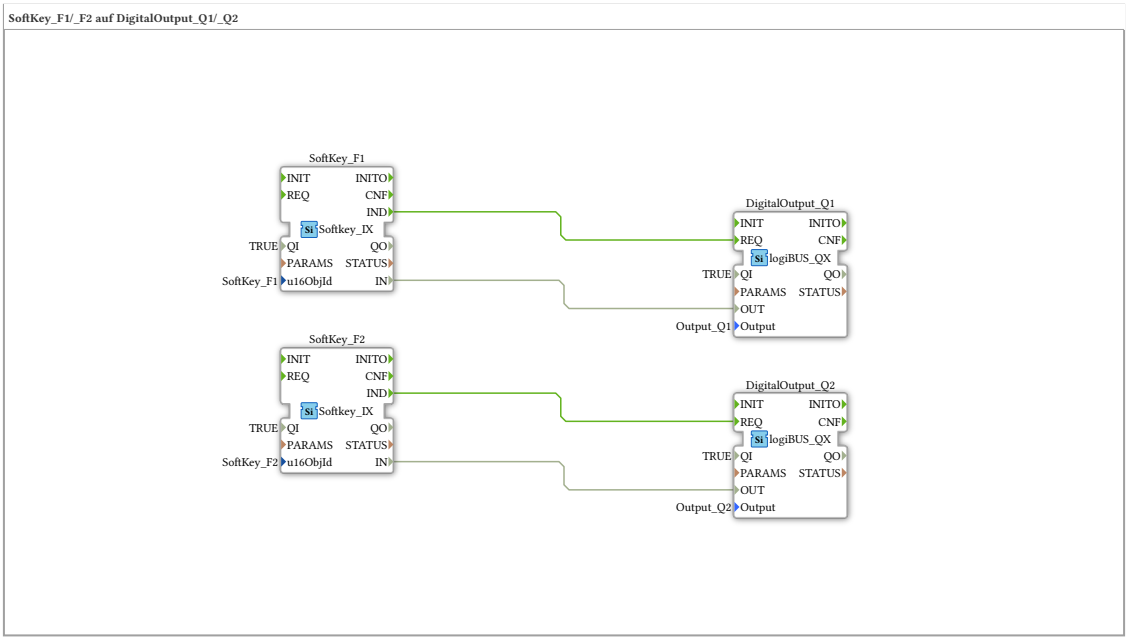
- 🌐 Interaktiver JBC-Lötspitzen Guide & Infografik auf ms-muc-docs.de

2.2.1.104 Uebung_010a: SoftKey_F1/_F2 auf DigitalOutput_Q1/_Q2

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_010a`.

🎧 Podcast



- ISO 11783-6: Softkeys und das Virtual Terminal verstehen – Dein Schlüssel zur Landmaschinen-Mechatronik

---

Ziel der Übung

Erweiterung der ISOBUS-Steuerung auf mehrere Kanäle.

Beschreibung und Komponenten

Die Subapplikation `Uebung_010a.SUB` steuert zwei unabhängige Hardware-Ausgänge über zwei Softkeys am Terminal.

Funktionsbausteine (FBs)

- **`SoftKey_F1`** ➡️ **`DigitalOutput_Q1`**
- **`SoftKey_F2`** ➡️ **`DigitalOutput_Q2`**

Beide Signalpfade nutzen die ereignisbasierte `IND -> REQ` Verbindung.

Funktionsweise

Dies demonstriert, dass das UT-Interface beliebig skaliert werden kann. Jeder Softkey im Objekt-Pool kann als eigenständige Instanz im 4diac-Programm genutzt werden, um spezifische Aktoren anzusteuern.

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

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

2.2.1.105 Uebung_010a2: Button_A1 auf DigitalOutput_Q1



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_010a2`. Neben den Softkeys am Rand gibt es im ISOBUS auch „Buttons“, die sich direkt innerhalb der Arbeitsmaske befinden.

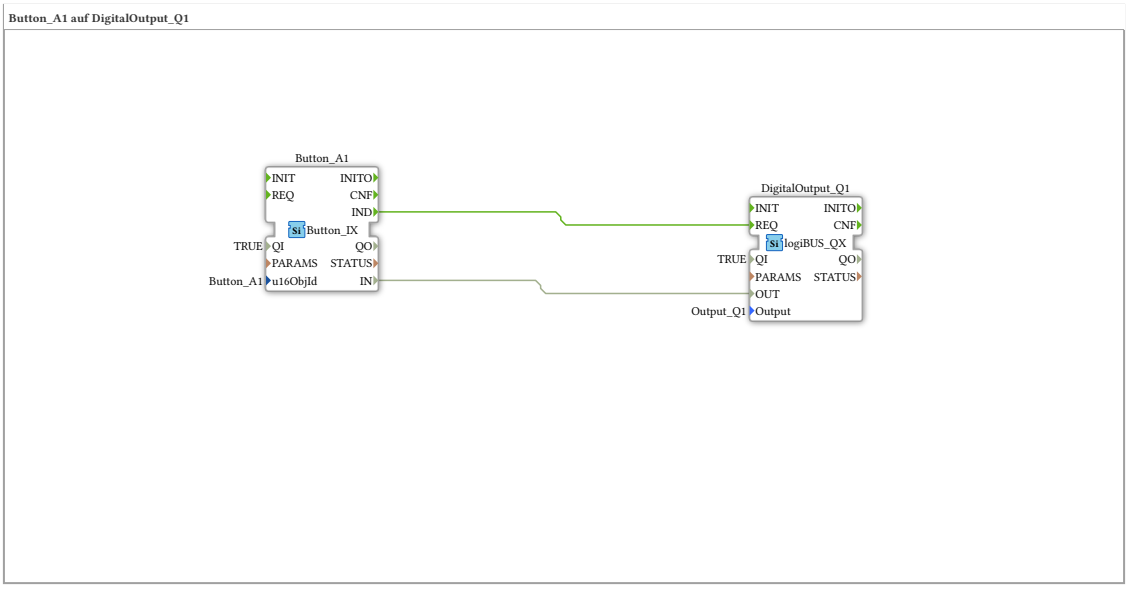
Ziel der Übung

Verwendung eines `Button_IX` Bausteins.

Beschreibung und Komponenten

Die Subapplikation `Uebung_010a2.SUB` nutzt einen Button anstelle eines Softkeys zur Steuerung eines Ausgangs.

Funktionsbausteine (FBs)



- **`Button_A1`**: Typ `isobus::UT::io::Button::Button_IX`. Referenziert das Objekt `Button_A1` im Pool.

Funktionsweise

Die Logik ist identisch zum Softkey: Solange die Schaltfläche auf dem Bildschirm berührt wird, liefert der Baustein `TRUE`. Der wesentliche Unterschied ist die visuelle Platzierung und Gestaltungsmöglichkeit innerhalb der grafischen Benutzeroberfläche des Terminals.

2.2.1.106 Uebung_010a3: Button_L1 (Latching) auf DigitalOutput_Q1



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_010a3`.

Ziel der Übung

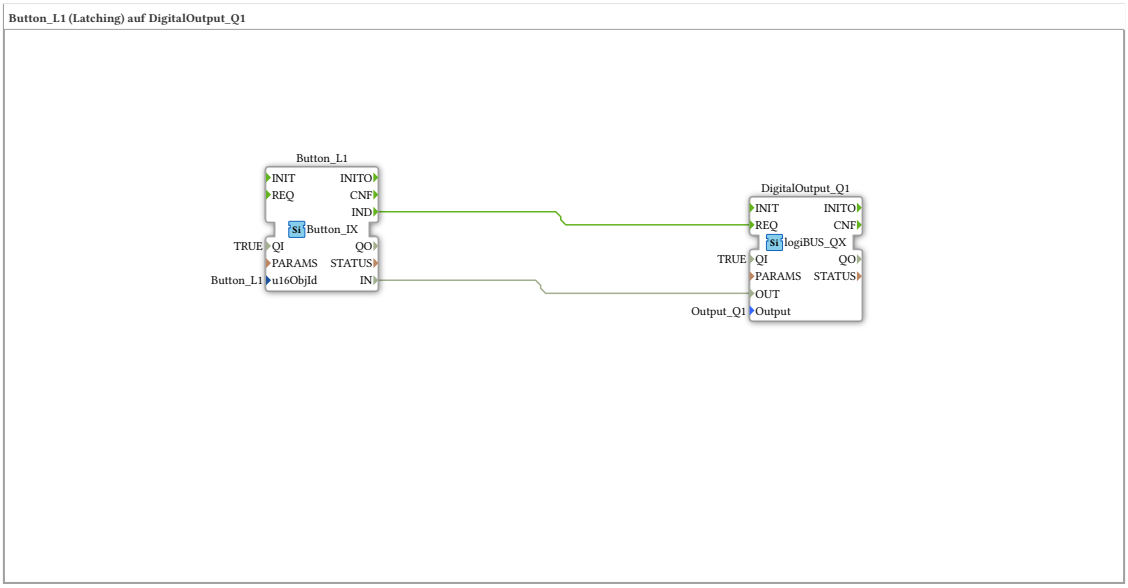
Umgang mit zustandshaltenden Bedienelementen des Universal Terminals.

Beschreibung und Komponenten

In `Uebung_010a3.SUB` wird ein `Button_L1` (Latching) verwendet.

Funktionsweise

Ein „Latching Button“ ist im ISOBUS-Objektpool so definiert, dass er seinen Zustand bei Betätigung speichert.



- Erster Klick: Button rastet visuell ein, sendet dauerhaft `TRUE`.
- Zweiter Klick: Button springt zurück, sendet dauerhaft `FALSE`.

Daher wird, wie im Kommentar vermerkt, **kein Software-Flip-Flop** (T_FF) in 4diac benötigt. Die Speicherfunktion wird vollständig vom ISOBUS-Terminal übernommen.

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

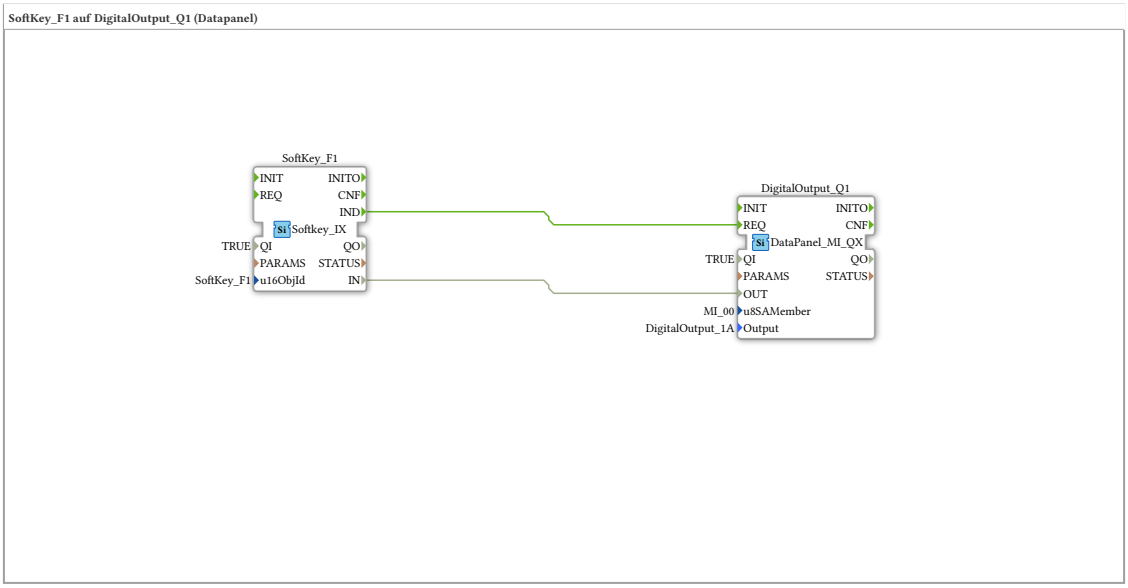
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.2.1.107 Uebung_010a4: SoftKey_F1 auf DigitalOutput_Q1 (Datapanel)



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_010a4`.

🎧 Podcast



- ISO 11783-6: Softkeys und das Virtual Terminal verstehen – Dein Schlüssel zur Landmaschinen-Mechatronik

---

Ziel der Übung

Kombination verschiedener logiBUS-Teilsysteme.

Beschreibung und Komponenten

Die Subapplikation `Uebung_010a4.SUB` verbindet einen ISOBUS-Softkey mit einem physikalischen Ausgang eines DataPanels.

Funktionsbausteine (FBs)

- **`SoftKey_F1`**: Eingabequelle vom Traktor-Terminal.
- **`DigitalOutput_Q1`**: Typ `DataPanel_MI_QX`. Dies ist ein Ausgang auf einer dezentralen IO-Box am CAN-Bus.

Funktionsweise

Diese Übung verdeutlicht die Mächtigkeit der IEC 61499 Abstraktion: Für die Programmlogik ist es völlig unerheblich, woher ein Signal kommt (Software-Terminal) oder wohin es geht (CAN-Modul). Die Ereignis- und Datenverbindungen überbrücken die Protokollgrenzen zwischen ISOBUS und gerätespezifischem CAN-Protokoll nahtlos.

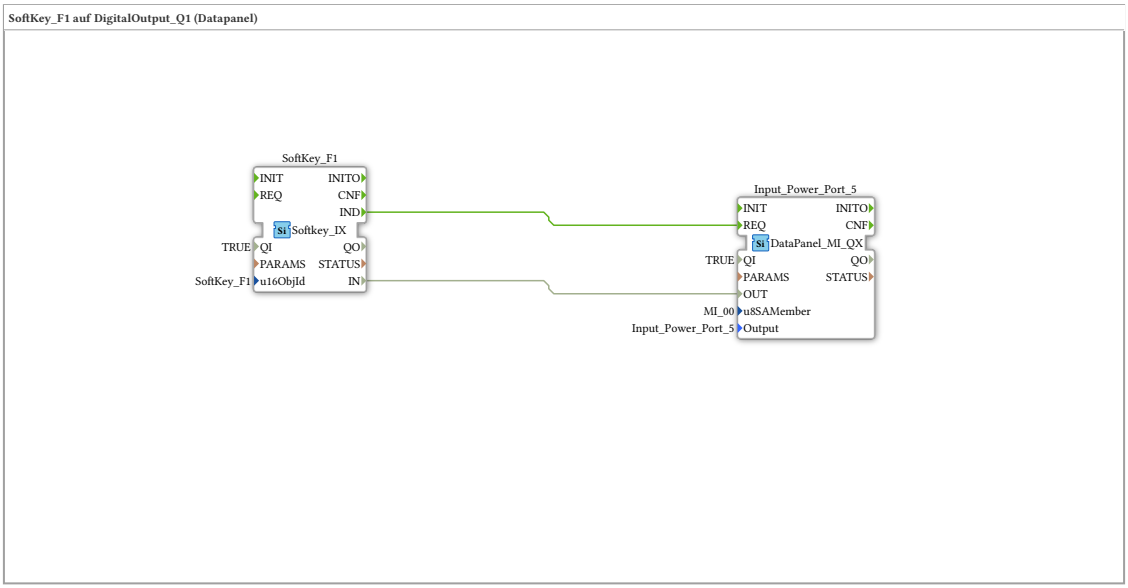
2.2.1.108 Uebung_010a5: SoftKey_F1 auf DigitalOutput_Q1 (Datapanel)

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_010a5`. Ein Softkey auf dem ISOBUS-Terminal schaltet einen DataPanel-Ausgang.

Ziel der Übung

Der Softkey `SoftKey_F1` auf dem VT-Terminal schaltet direkt einen digitalen Ausgang auf dem DataPanel-Erweiterungsmodul.

Beschreibung und Komponenten



- **`SoftKey_F1`**: `isobus::UT::io::Softkey::Softkey_IX`, liest den Zustand des Softkeys `SoftKey_F1` vom VT.
- **`Input_Power_Port_5`**: `DataPanel::io::MI::DQ::DataPanel_MI_QX`, digitaler Ausgang auf dem DataPanel (`u8SAMember = MI_00`).

Funktionsweise

1. Der Bediener betätigt `SoftKey_F1` auf dem VT-Terminal.
2. Der Zustand des Softkeys (`SoftKey_F1.IN`) wird direkt an den DataPanel-Ausgang (`Input_Power_Port_5.OUT`) durchgereicht.
3. Jede Zustandsänderung (`IND`) löst eine Anforderung (`REQ`) am Ausgang aus.

2.2.1.109 Uebung_010b1: AuxFunction2_X1 auf DigitalOutput_Q1

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_010b1`. Hier wird die dritte Säule der ISOBUS-Bedienung eingeführt: Auxiliary Functions (AUX-N).

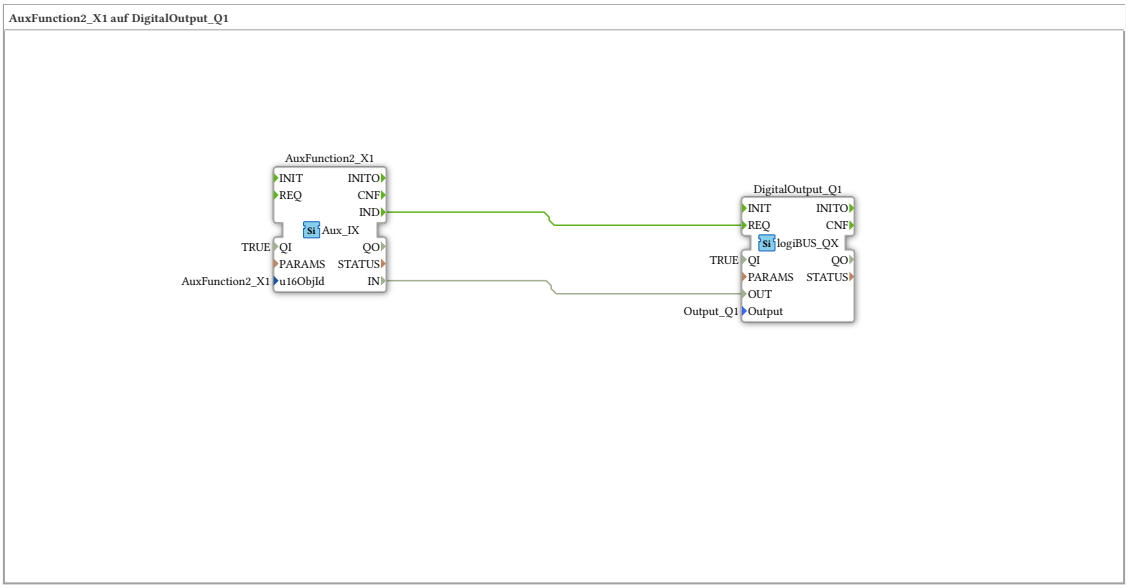
Ziel der Übung

Anbindung von AUX-Eingabegeräten (z.B. ISOBUS-Joystick).

Beschreibung und Komponenten

In `Uebung_010b1.SUB` wird eine Auxiliary Function genutzt, um einen Ausgang zu schalten.

Funktionsbausteine (FBs)



- **`AuxFunction2_X1`**: Typ `isobus::UT::io::Auxiliary::IN::Aux_IX`. Dieser Baustein lauscht auf AUX-Nachrichten der „Funktion 2“.

Funktionsweise

Im Gegensatz zu Softkeys, die ein festes Bildschirmelement sind, ist eine AUX-Funktion ein logisches Objekt. Der Bediener muss am Terminal (über das AUX-Menü) einmalig festlegen, welche physische Taste seines Joysticks er dieser „Funktion 2“ zuweisen möchte. Sobald dieses „Teaching“ abgeschlossen ist, triggert jeder Druck auf die Joystick-Taste den Baustein in 4diac.

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

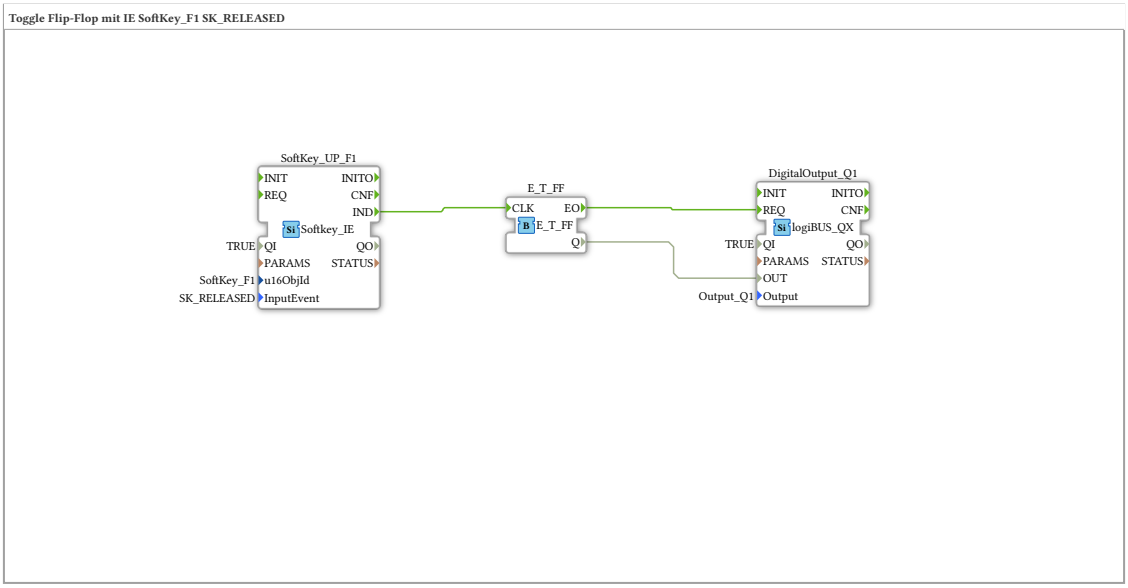
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.2.1.110 Uebung_010b2: Toggle Flip-Flop mit IE SoftKey_F1 SK_RELEASED



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_010b2`.

Podcast



- ISO 11783-6: Softkeys und das Virtual Terminal verstehen – Dein Schlüssel zur Landmaschinen-Mechatronik

---

Ziel der Übung

Verwendung spezialisierter ISOBUS-Ereignisse zur Steuerung von Software-Flip-Flops.

Beschreibung und Komponenten

Die Subapplikation `Uebung_010b2.SUB` nutzt ein Flip-Flop, das durch das Loslassen eines Softkeys getriggert wird.

Funktionsbausteine (FBs)

- **`SoftKey_UP_F1`**: Typ `isobus::UT::io::Softkey::Softkey_IE`. Er ist auf das Ereignis `SK_RELEASED` konfiguriert.
- **`E_T_FF`**: Toggle-Flip-Flop.

Funktionsweise

Das Ereignis `IND` wird erst gefeuert, wenn der Nutzer den Finger vom Softkey nimmt. Dies entspricht dem intuitiven Klick-Verhalten. Jeder vollständige Tastendruck (Drücken + Loslassen) schaltet das Licht um.

2.2.1.111 Uebung_010b3: Toggle Flip-Flop mit IE AuxFunction2_X1

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_010b3`.

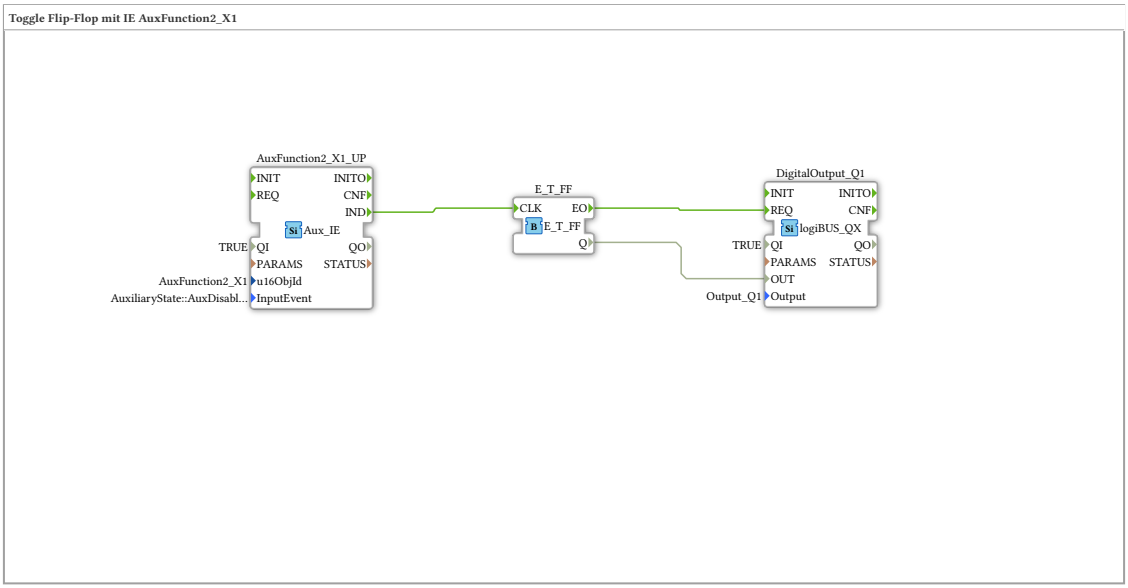
Ziel der Übung

Verwendung von `Aux_IE` (Event) zur Steuerung von Speichern.

Beschreibung

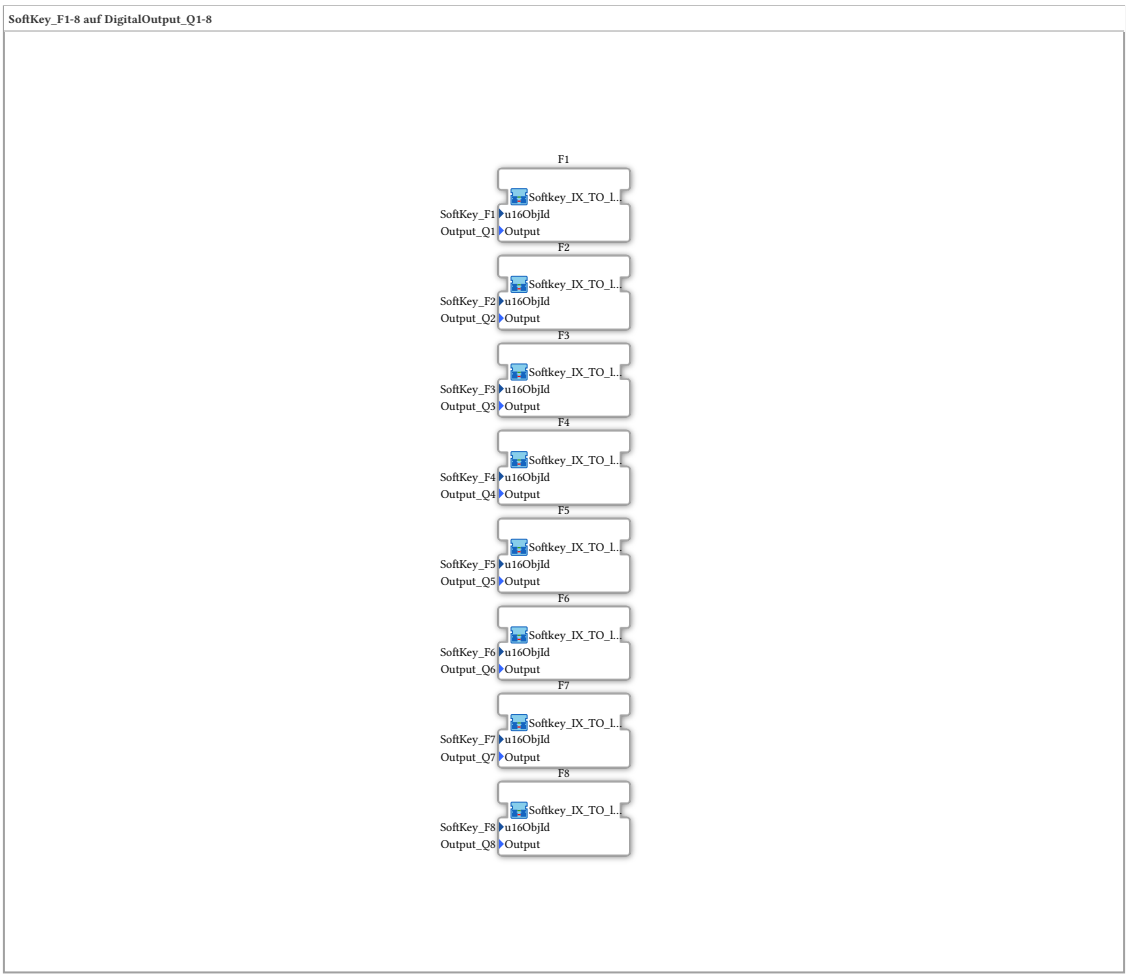
In `Uebung_010b3.SUB` wird eine AUX-Funktion genutzt, um ein Flip-Flop zu toggeln.

Funktionsweise



Es wird das Event `AuxDisabled_START` verwendet. In der ISOBUS-Terminologie bedeutet dies den Übergang in den Zustand „Deaktiviert“. Das entspricht dem **Loslassen** einer Joystick-Taste. Das Flip-Flop wechselt also beim Loslassen der Taste seinen Zustand.

2.2.1.112 Uebung_010b4: SoftKey_F1-8 auf DigitalOutput_Q1-8



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_010b4`.

🎧 Podcast

- ISO 11783-6: Softkeys und das Virtual Terminal verstehen – Dein Schlüssel zur Landmaschinen-Mechatronik

---

Übersicht

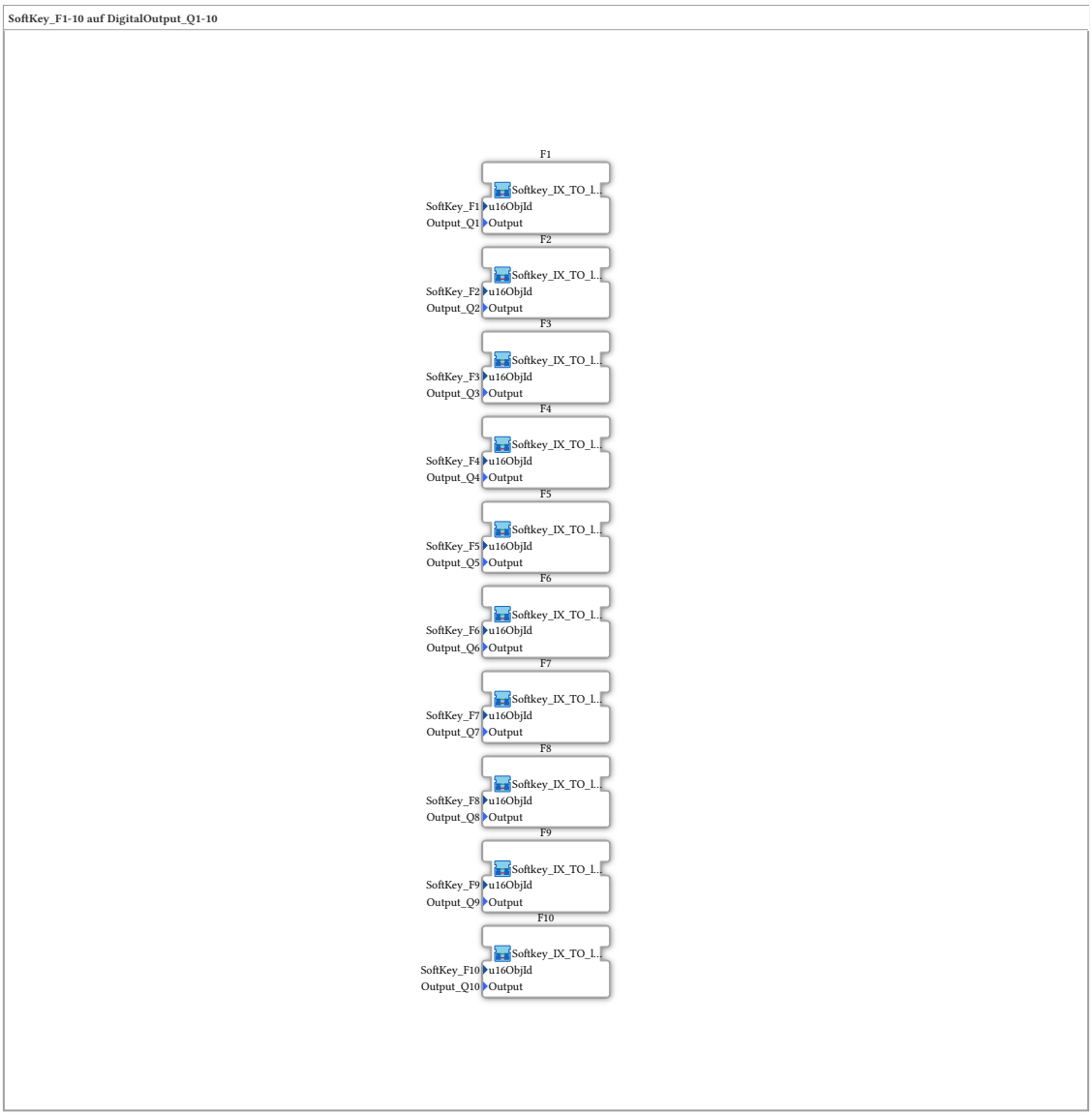
Diese Übung demonstriert das Massen-Mapping von 8 ISOBUS-Softkeys auf 8 Hardware-Ausgänge unter Verwendung von untypisierten Sub-Applikationen (`Uebung_010b4_sub`). Dies sorgt für ein extrem aufgeräumtes Hauptdiagramm bei gleichzeitig hoher Kanalanzahl.

2.2.1.113 Uebung_010b4_sub

Hinweis

Diese Übung existiert in den aktuellen Trainingsquellen nicht mehr (entfernt, zusammengelegt oder umbenannt). Diese Seite bleibt nur als Platzhalter für bestehende Verweise erhalten. Die aktuellen Übungen stehen in der Übungsübersicht.

2.2.1.114 Uebung_010b5: SoftKey_F1-10 auf DigitalOutput_Q1-10



NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_010b5`.

🎧 Podcast

- ISO 11783-6: Softkeys und das Virtual Terminal verstehen – Dein Schlüssel zur Landmaschinen-Mechatronik

---

Übersicht

Erweiterung der vorherigen Übung auf insgesamt 10 parallele Softkey-Kanäle. Jede Sub-Applikation kapselt die Logik für genau einen Softkey (F1 bis F10).

2.2.1.115 Uebung_010b5_sub

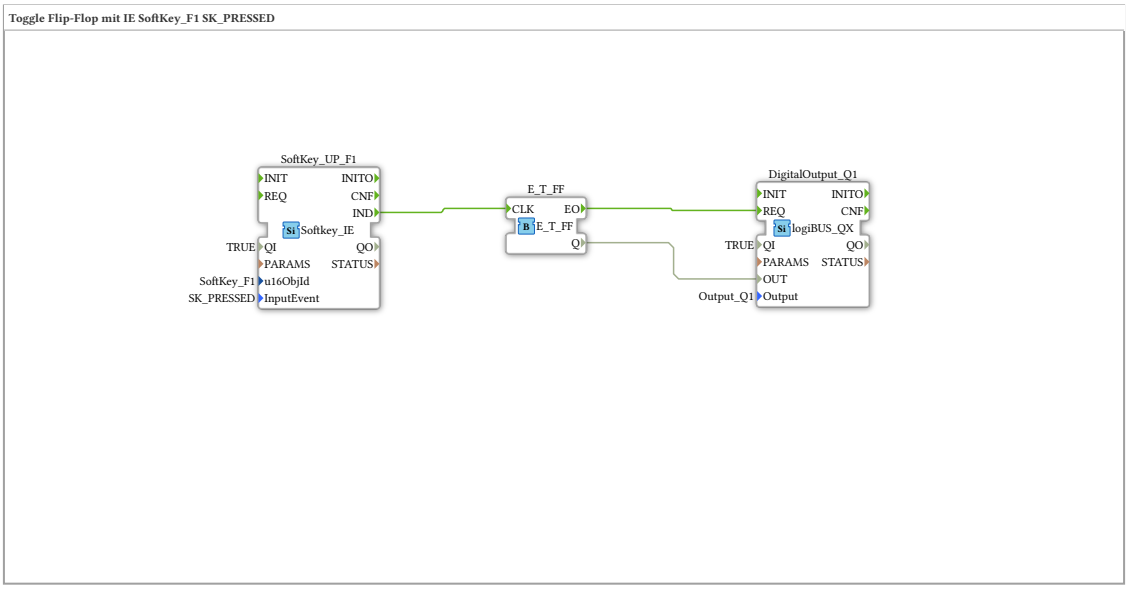
Hinweis

Diese Übung existiert in den aktuellen Trainingsquellen nicht mehr (entfernt, zusammengelegt oder umbenannt). Diese Seite bleibt nur als Platzhalter für bestehende Verweise erhalten. Die aktuellen Übungen stehen in der Übungsübersicht.

2.2.1.116 Uebung_010b6: Toggle Flip-Flop mit IE SoftKey_F1 SK_PRESSED

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_010b6`.

Podcast



- ISO 11783-6: Softkeys und das Virtual Terminal verstehen – Dein Schlüssel zur Landmaschinen-Mechatronik

---

Ziel der Übung

Reaktion zum frühestmöglichen Zeitpunkt der Interaktion.

Funktionsweise

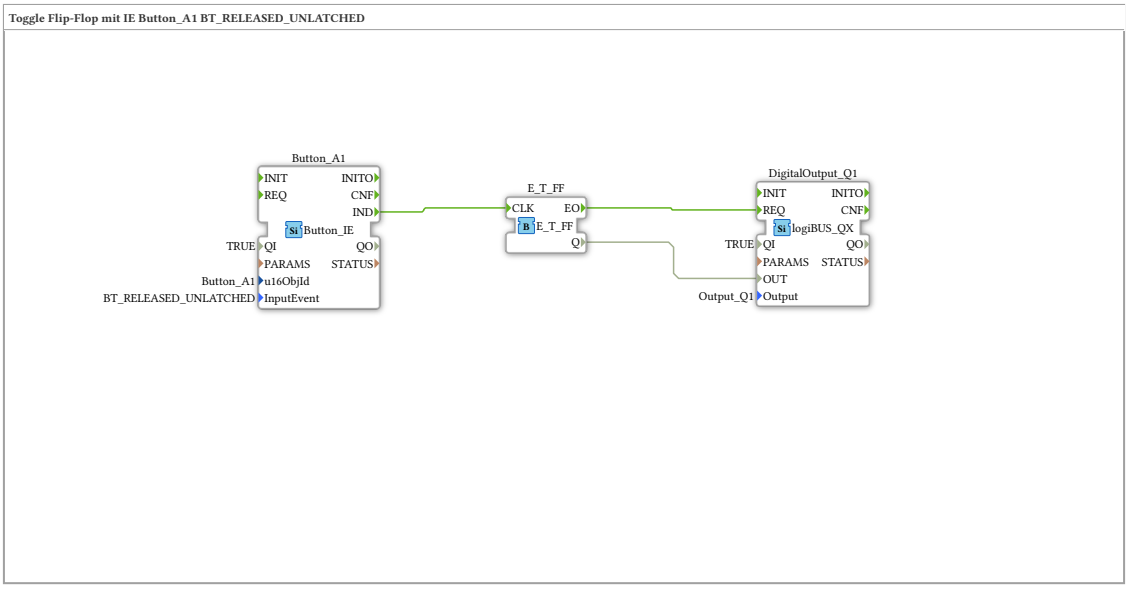
Verwendet das Event `SK_PRESSED`. Das Flip-Flop am Ausgang toggelt bereits in dem Moment, in dem der Nutzer den Touchscreen berührt. Dies minimiert die gefühlte Latenz, verhindert aber ein nachträgliches Abbrechen durch Wegziehen des Fingers.

2.2.1.117 Uebung_010b7: Toggle Flip-Flop mit IE Button_A1 BT_RELEASED_UNLATCHED

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_010b7`.

Funktionsweise

Nutzt `Button_A1` mit dem Ereignis `BT_RELEASED_UNLATCHED`. Das Event wird gefeuert, wenn ein normaler (nicht-rastender) Button auf der Arbeitsmaske losgelassen wird. Dies entspricht einem klassischen Mausklick-Ereignis.



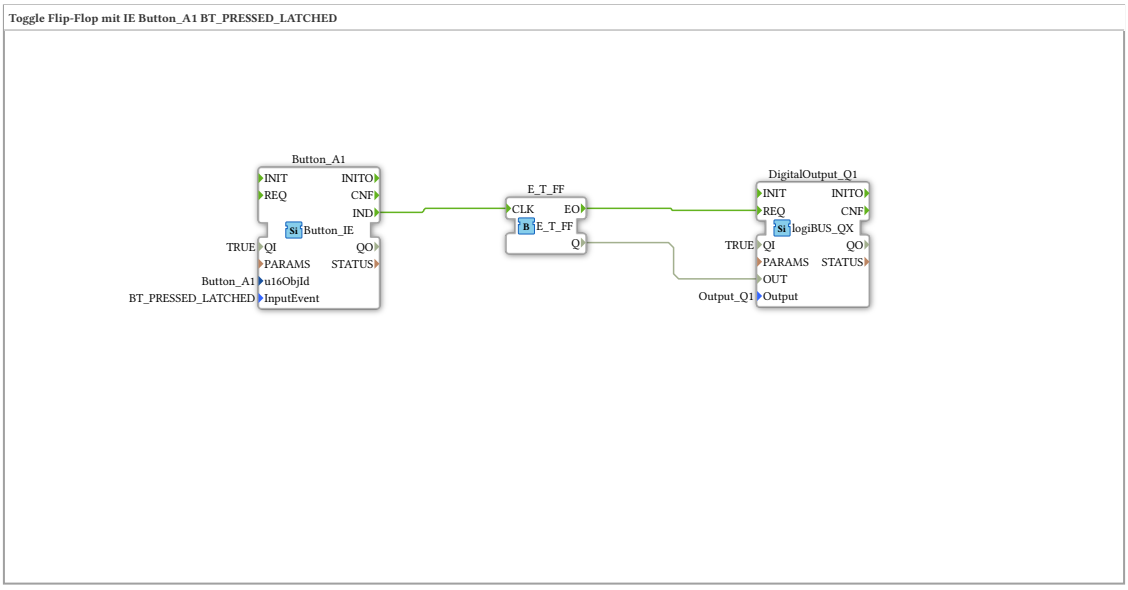
2.2.1.118 Uebung_010b8: Toggle Flip-Flop mit IE Button_A1 BT_PRESSED_LATCHED

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_010b8`.

Funktionsweise

Nutzt `Button_A1` mit `BT_PRESSED_LATCHED`. Dieses Ereignis ist speziell für Buttons gedacht, die visuell einrasten sollen. Das Ereignis wird ausgelöst, sobald der Button in den Zustand „Aktiviert/Eingerastet“ wechselt.



2.2.1.119 Uebung_010b9: Toggle Flip-Flop mit IE Button_A1 BT_STILL_HELD

NotebookLM

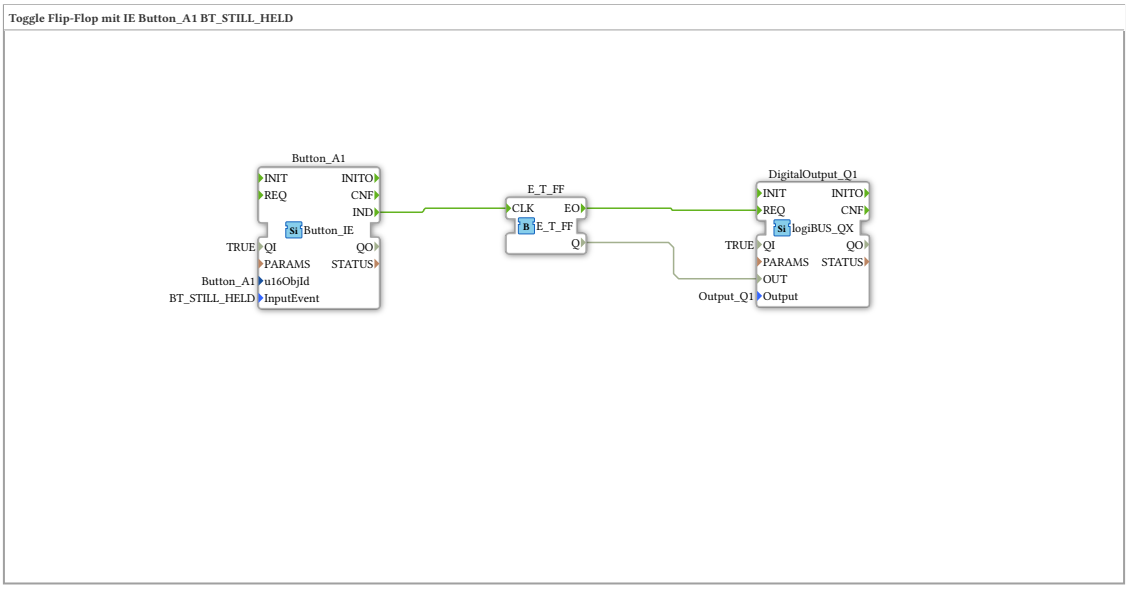
Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_010b9`.

Ziel der Übung

Nutzung repetierender Ereignisse zur Erzeugung von Blinksignalen oder Inkrement-Funktionen.

Funktionsweise

Nutzt `Button_A1` mit dem Ereignis `BT_STILL_HELD`. Wie im Kommentar vermerkt, wird dieses Ereignis alle 200ms wiederholt, solange der Finger auf dem Button bleibt. Da das Signal an ein Toggle-Flip-Flop geleitet wird, blinkt der Hardware-Ausgang mit einer Periode von 400ms (200ms an, 200ms aus), solange gedrückt wird.

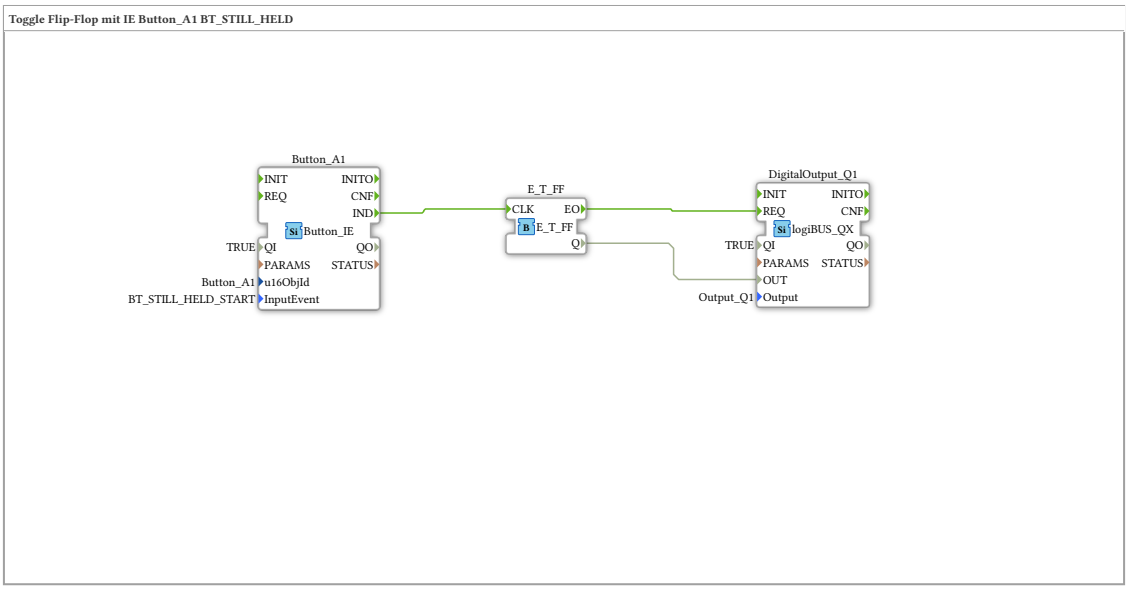


2.2.1.120 Uebung_010bA: Toggle Flip-Flop mit IE Button_A1 BT_STILL_HELD



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_010bA`.

Funktionsweise



Nutzt `Button_A1` mit `BT_STILL_HELD_START`. Im Gegensatz zum einfachen `STILL_HELD` wird dieses Ereignis **nicht wiederholt**. Es feuert exakt einmal, sobald die Haltezeit überschritten wurde. Dies entspricht einer sauberen „Long Press“ Auswertung.

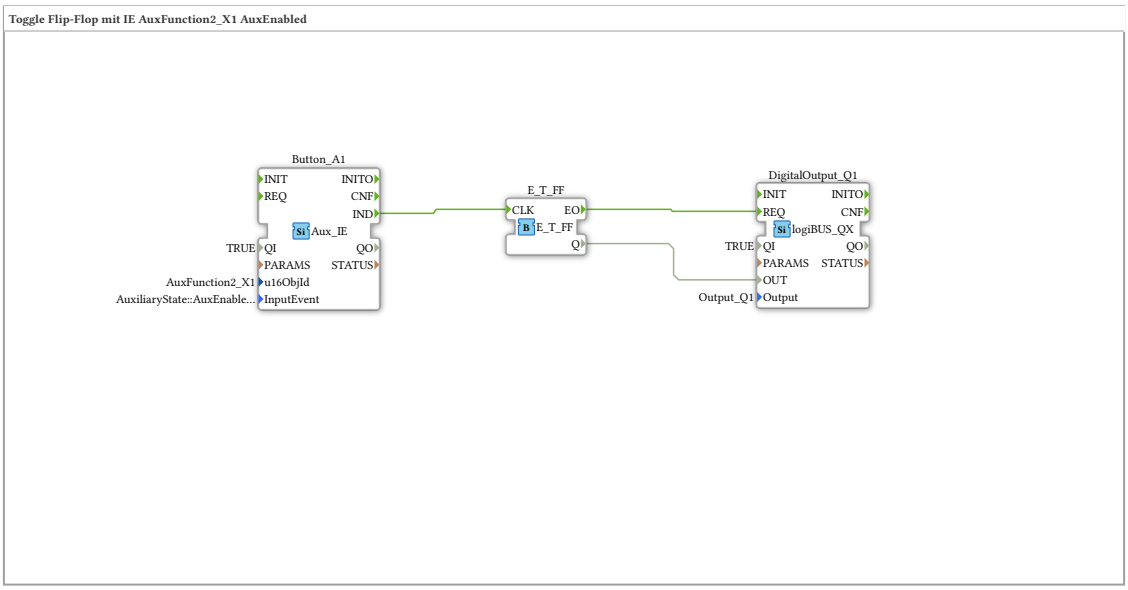
2.2.1.121 Uebung_010bA2: Toggle Flip-Flop mit IE AuxFunction2_X1 AuxEnabled



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_010bA2`. Hier geht es um die Feinheiten der AUX-Spezifikation bezüglich rastender und tastender Eingänge.

Funktionsweise

Verwendet `AuxFunction2_X1` mit dem Event `AuxEnabled`. Das Verhalten hängt vom Typ des zugewiesenen Bedienelements (Joystick-Taste) ab:



- Bei einem **tastenden Bediener** (NonLatched) wird das Event nur einmal beim Drücken gesendet.
- Bei einem **rastenden Bediener** (Latched) wird das Event zyklisch wiederholt, solange der Schalter aktiv ist. Dies verdeutlicht, wie die Software auf die Hardware-Eigenschaften des verwendeten Joysticks reagiert.

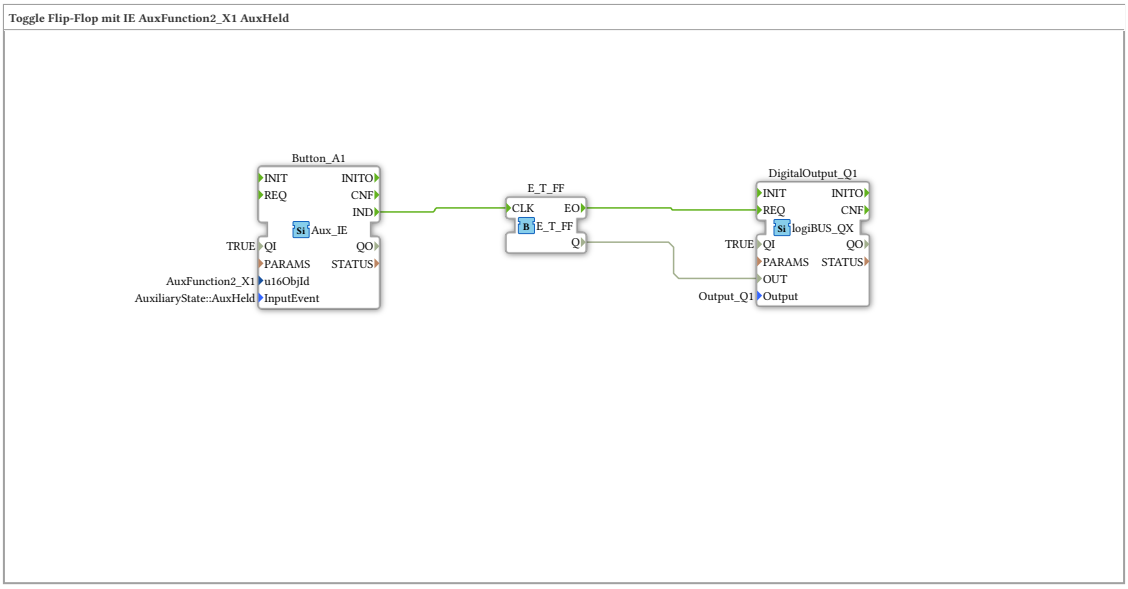
2.2.1.122 Uebung_010bA3: Toggle Flip-Flop mit IE AuxFunction2_X1 AuxHeld



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_010bA3`.

Funktionsweise

Nutzt `AuxFunction2_X1` mit `AuxHeld`. Bei einem tastenden Bedienelement führt dieses Ereignis zu einer kontinuierlichen Event-Folge, solange die Taste am Joystick gehalten wird. In Kombination mit einem Toggle-Flip-Flop entsteht so ein Blink-Effekt am Ausgang.

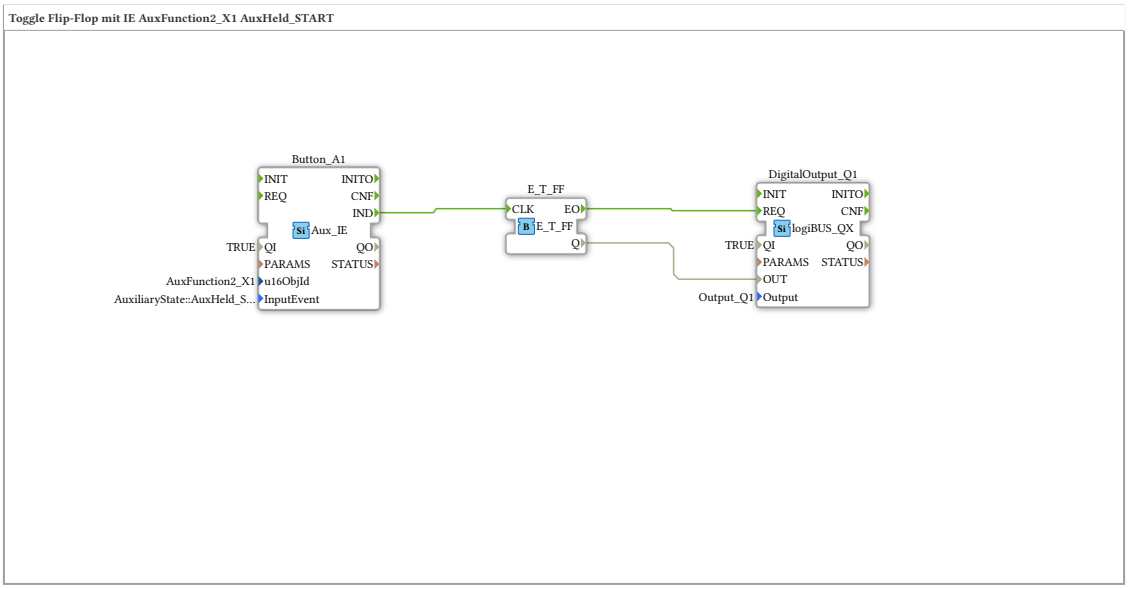


2.2.1.123 Uebung_010bA4: Toggle Flip-Flop mit IE AuxFunction2_X1 AuxHeld_START

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_010bA4`.

Funktionsweise



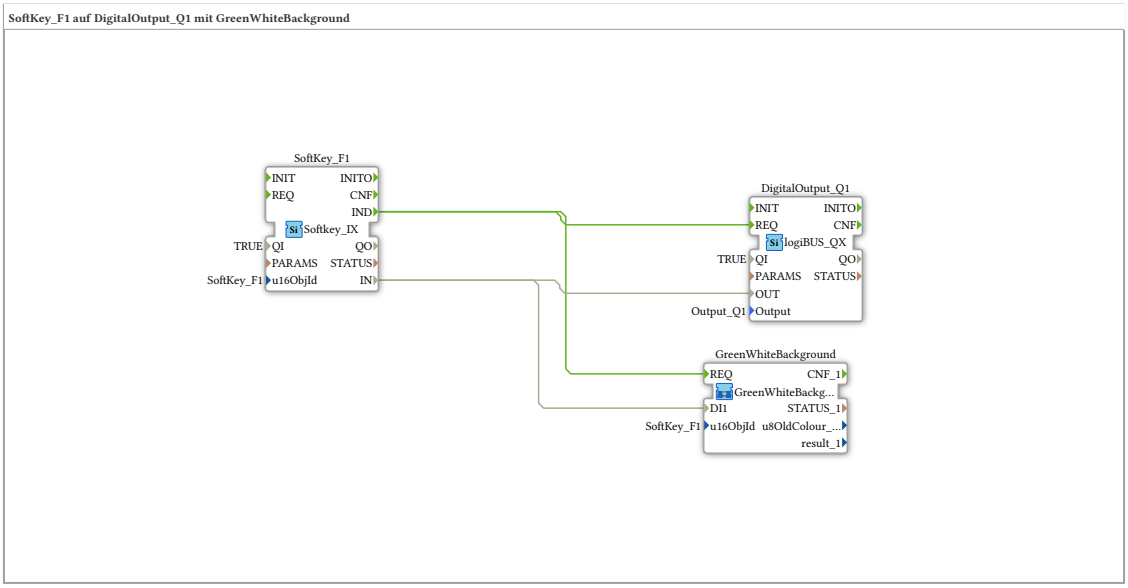
Nutzt `AuxFunction2_X1` mit `AuxHeld_START`. Unabhängig vom Typ des Bedienelements wird dieses Ereignis nur **einmal** beim Erreichen der Zeitschwelle gesendet. Es ist die bevorzugte Wahl für Long-Press-Funktionen an ISOBUS-Joysticks.

2.2.1.124 Uebung_010c: SoftKey_F1 auf DigitalOutput_Q1 mit GreenWhiteBackground

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_010c`. Bisher dienten die Tasten nur der Eingabe. Jetzt erhalten sie eine dynamische Rückmeldung auf dem Bildschirm.

Podcast



- ISO 11783-6: Softkeys und das Virtual Terminal verstehen – Dein Schlüssel zur Landmaschinen-Mechatronik

---

Ziel der Übung

Rückmeldung an den Bediener durch Farbumschlag der virtuellen Taste.

Beschreibung und Komponenten

Die Subapplikation `Uebung_010c.SUB` erweitert die einfache Schaltung um einen Feedback-Baustein.

Funktionsbausteine (FBs)

- **`SoftKey_F1`**: Eingabe-Baustein.
- **`GreenWhiteBackground` (SubApp)**: Ein Baustein aus der Bibliothek `MyLib::sys`. Er sorgt dafür, dass sich der Hintergrund des Softkeys auf dem Terminal ändert Grün bei Aktivierung, Weiß im Ruhezustand.
- **`DigitalOutput_Q1`**: Der physische Ausgang.

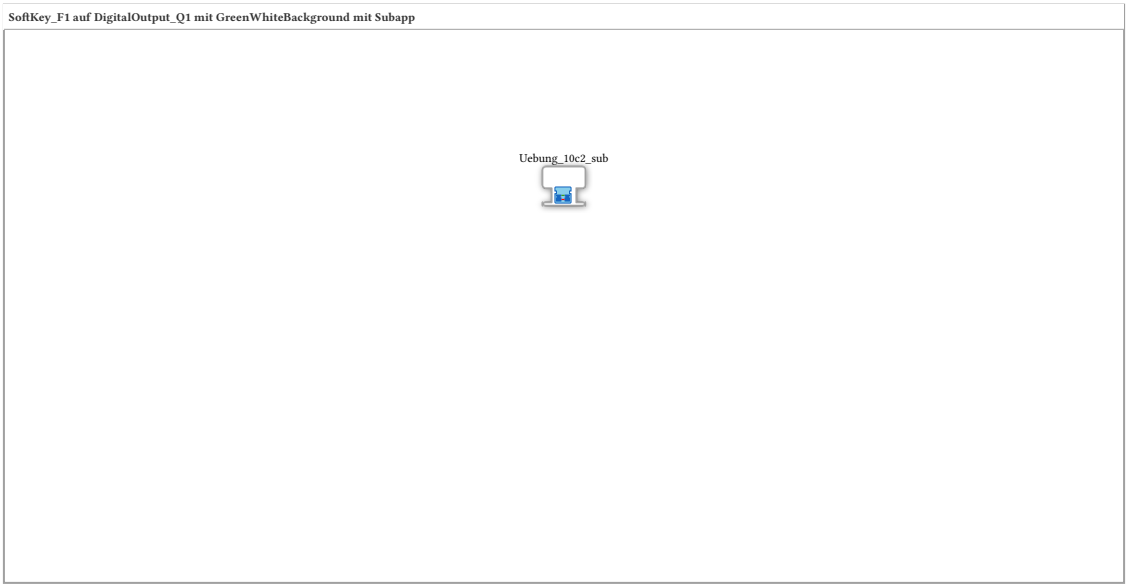
Funktionsweise

Das Signal vom Softkey wird aufgeteilt:

1. Ein Zweig geht zum physischen Ausgang `Q1`.
2. Der zweite Zweig geht zum Feedback-Baustein.

Drückt der Nutzer die Taste, leuchtet nicht nur die Lampe an der Maschine, sondern die Taste auf dem Terminal-Bildschirm wird gleichzeitig grün hervorgehoben. Dies gibt dem Nutzer die Sicherheit, dass sein Befehl vom System registriert wurde.

2.2.1.125 Uebung_010c2: SoftKey_F1 auf DigitalOutput_Q1 mit GreenWhiteBackground mit Subapp



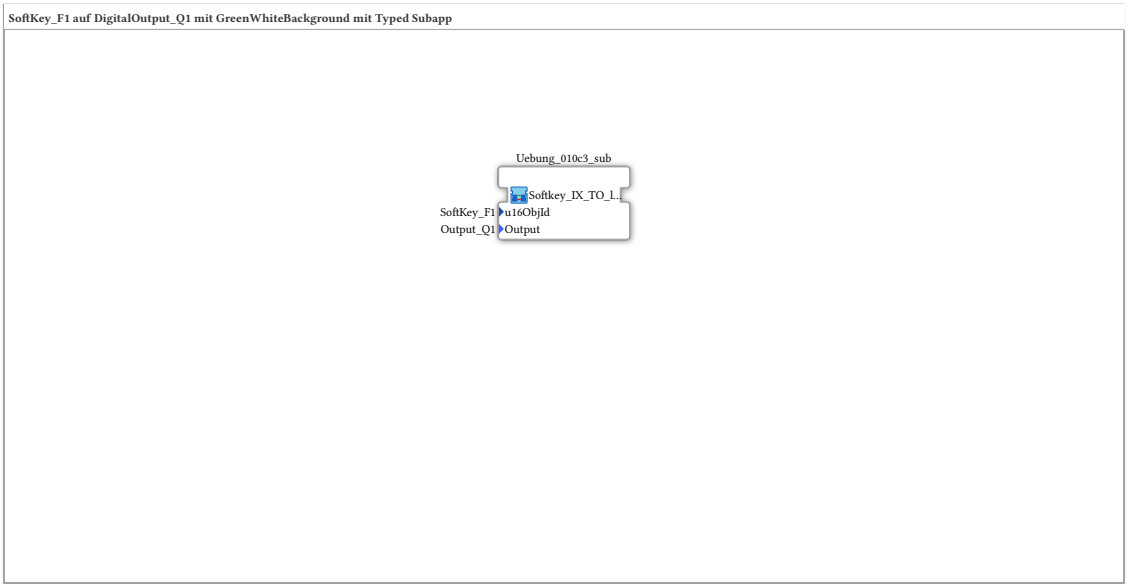
NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_010c2`.

Übersicht

Die komplette Feedback-Logik aus der vorherigen Übung wird in eine untypisierte Sub-Applikation `Uebung_10c2_sub` gekapselt. Dies zeigt, wie man komplexe Interaktions-Logiken (Eingabe + Feedback + Hardware-Ausgabe) visuell gruppiert, um die Hauptansicht des Programms übersichtlich zu halten.

2.2.1.126 Uebung_010c3: SoftKey_F1 auf DigitalOutput_Q1 mit GreenWhiteBackground mit Typed Subapp



NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_010c3`.

Übersicht

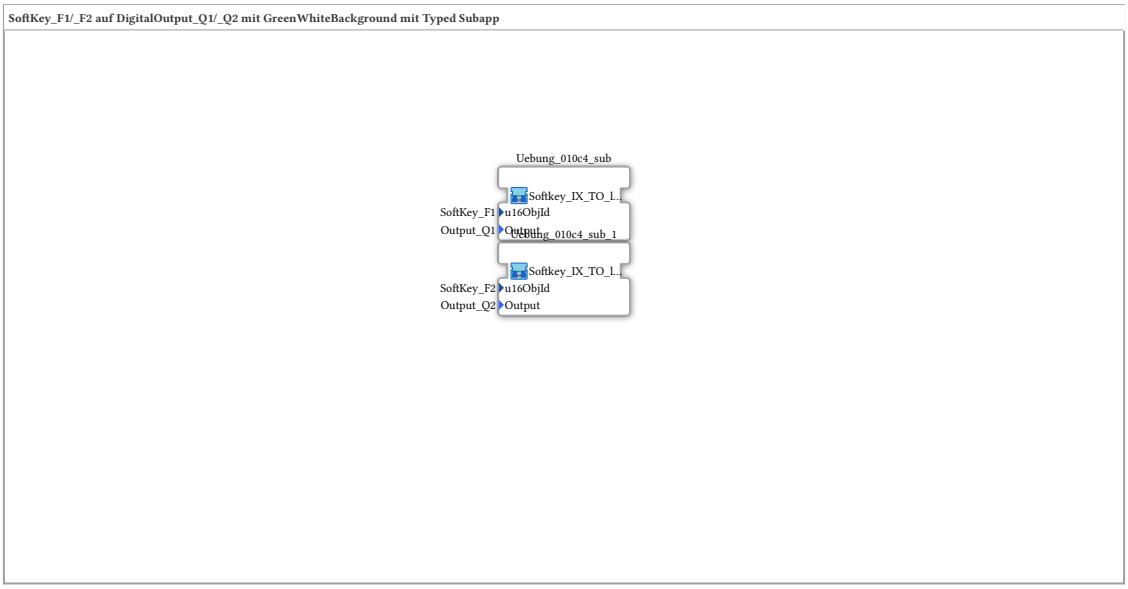
Hier wird die Kapselung konsequent zu Ende geführt: Aus der Sub-Applikation wird ein echter Typ `Uebung_010c3_sub` erstellt. Dieser Baustein kann nun wie ein Standard-Baustein aus der Bibliothek verwendet werden. Er verlangt lediglich die Angabe der Softkey-ID und des Hardware-Ausgangs als Parameter und erledigt die gesamte interne Verschaltung von Logik und Feedback automatisch.

2.2.1.127 Uebung_010c3_sub

Hinweis

Diese Übung existiert in den aktuellen Trainingsquellen nicht mehr (entfernt, zusammengelegt oder umbenannt). Diese Seite bleibt nur als Platzhalter für bestehende Verweise erhalten. Die aktuellen Übungen stehen in der Übungsübersicht.

2.2.1.128 Uebung_010c4: SoftKey_F1/_F2 auf DigitalOutput_Q1/_Q2 mit GreenWhite-Background mit Typed Subapp



NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_010c4`.

Übersicht

Beweis der Skalierbarkeit: Der in Übung 010c3 erstellte Typ wird zweimal instanziiert, um zwei Softkeys (`F1`, `F2`) mit zwei Ausgängen (`Q1`, `Q2`) und jeweiligem individuellen Bildschirm-Feedback zu verknüpfen. Dies demonstriert, wie effizient komplexe Maschinen-Interfaces mit IEC 61499 Bausteinen aufgebaut werden können.

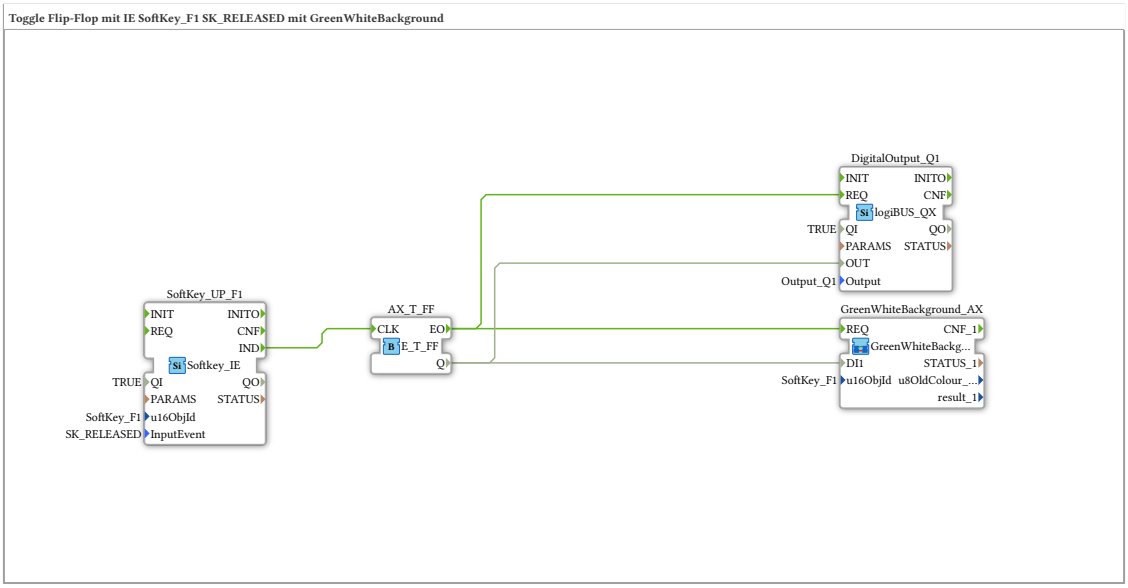
2.2.1.129 Uebung_010c4_sub

Hinweis

Diese Übung existiert in den aktuellen Trainingsquellen nicht mehr (entfernt, zusammengelegt oder umbenannt). Diese Seite bleibt nur als Platzhalter für bestehende Verweise erhalten. Die aktuellen Übungen stehen in der Übungsübersicht.

2.2.1.130 Uebung_010d: Toggle Flip-Flop mit IE SoftKey_F1 SK_RELEASED mit Green-WhiteBackground

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_010d (Toggle Flip-Flop mit IE SoftKey_F1 SK_RELEASED mit GreenWhiteBackground).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Toggle Flip-Flop mit IE SoftKey_F1 SK_RELEASED mit GreenWhiteBackground**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_010d.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QX.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
- **SoftKey_UP_F1**: Instanz des Typs isobus::UT::io::Softkey::Softkey_IE.
  - Parameter QI = TRUE
  - Parameter u16ObjId = SoftKey_F1
  - Parameter InputEvent = SK_RELEASED
- **AX_T_FF**: Instanz des Typs iec61499::events::E_T_FF.

Verbindungen und Schnittstellen

Ereignisverbindungen:

- SoftKey_UP_F1.IND -> AX_T_FF.CLK
- AX_T_FF.EO -> DigitalOutput_Q1.REQ
- AX_T_FF.EO -> GreenWhiteBackground_AX.REQ

Datenverbindungen:

- AX_T_FF.Q -> DigitalOutput_Q1.OUT
- AX_T_FF.Q -> GreenWhiteBackground_AX.DI1

Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.

3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_010d demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

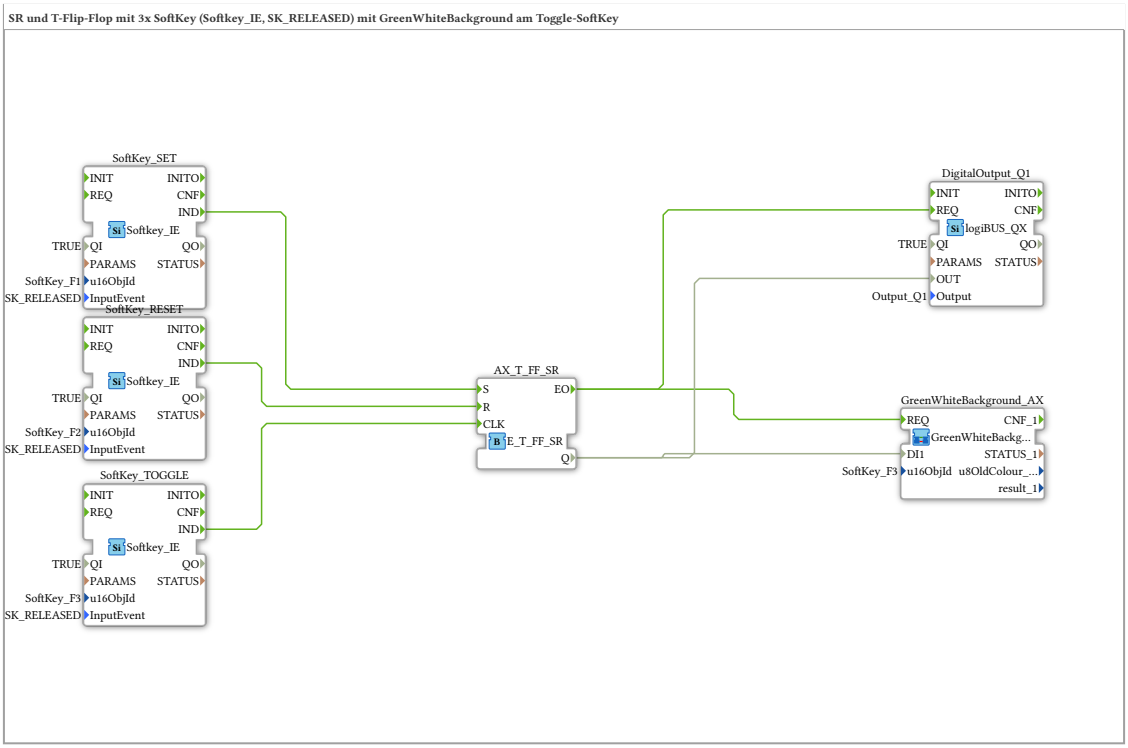
2.2.1.131 Uebung_010d2

Hinweis

Diese Übung existiert in den aktuellen Trainingsquellen nicht mehr (entfernt, zusammengelegt oder umbenannt). Diese Seite bleibt nur als Platzhalter für bestehende Verweise erhalten. Die aktuellen Übungen stehen in der Übungsübersicht.

2.2.1.132 Uebung_010e: SR und T-Flip-Flop mit 3x SoftKey (Softkey_IE, SK_RELEASED) mit GreenWhiteBackground am Toggle-SoftKey

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_010e (SR und T-Flip-Flop mit 3x SoftKey (Softkey_IE, SK_RELEASED) mit GreenWhiteBackground am Toggle-SoftKey).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **SR und T-Flip-Flop mit 3x SoftKey (Softkey_IE, SK_RELEASED) mit GreenWhiteBackground am Toggle-SoftKey**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_010e.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QX.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
- **SoftKey_SET**: Instanz des Typs isobus::UT::io::Softkey::Softkey_IE.
  - Parameter QI = TRUE

  - Parameter u16ObjId = SoftKey_F1
  - Parameter InputEvent = SK_RELEASED
- **SoftKey_RESET**: Instanz des Typs isobus::UT::io::Softkey::Softkey_IE.
  - Parameter QI = TRUE
  - Parameter u16ObjId = SoftKey_F2
  - Parameter InputEvent = SK_RELEASED
- **SoftKey_TOGGLE**: Instanz des Typs isobus::UT::io::Softkey::Softkey_IE.
  - Parameter QI = TRUE
  - Parameter u16ObjId = SoftKey_F3
  - Parameter InputEvent = SK_RELEASED
- **AX_T_FF_SR**: Instanz des Typs iec61499::events::E_T_FF_SR.

Verbindungen und Schnittstellen

Ereignisverbindungen:

- SoftKey_SET.IND -> AX_T_FF_SR.S
- SoftKey_RESET.IND -> AX_T_FF_SR.R
- SoftKey_TOGGLE.IND -> AX_T_FF_SR.CLK
- AX_T_FF_SR.EO -> DigitalOutput_Q1.REQ
- AX_T_FF_SR.EO -> GreenWhiteBackground_AX.REQ

Datenverbindungen:

- AX_T_FF_SR.Q -> DigitalOutput_Q1.OUT
- AX_T_FF_SR.Q -> GreenWhiteBackground_AX.DI1

Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

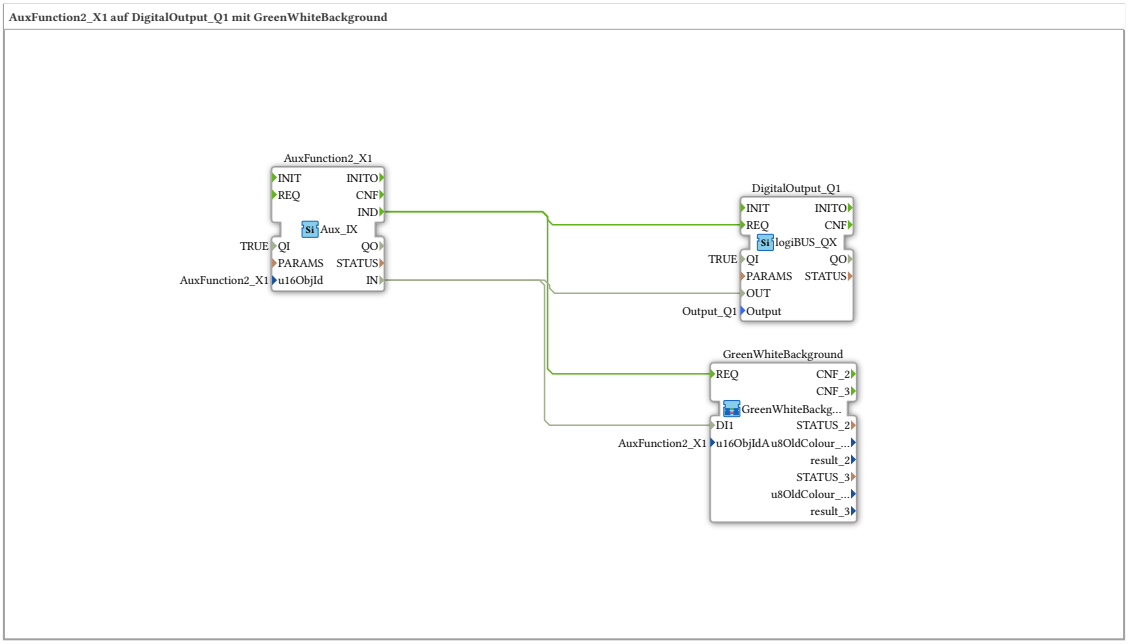
Die Übung Uebung_010e demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.2.1.133 Uebung_010e2

Hinweis

Diese Übung existiert in den aktuellen Trainingsquellen nicht mehr (entfernt, zusammengelegt oder umbenannt). Diese Seite bleibt nur als Platzhalter für bestehende Verweise erhalten. Die aktuellen Übungen stehen in der Übungsübersicht.

2.2.1.134 Uebung_010f: AuxFunction2_X1 auf DigitalOutput_Q1 mit GreenWhiteBackground



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung ist die klassische (nicht über AX-Adapter verdrahtete) Variante von `Uebung_010f_AX`, aufbauend auf `Uebung_010b1`: Zusätzlich zur Ansteuerung von `DigitalOutput_Q1` wird der Zustand von `AuxFunction2_X1` als Hintergrundfarbe (grün = EIN, weiß = AUS) zurückgemeldet.

Verwendete Funktionsbausteine (FBs)

- **AuxFunction2_X1**: Auxiliary-Function-Eingang (Typ: `isobus::UT::io::Auxiliary::IN::Aux_IX`)
  - **Parameter**: `QI = TRUE`, `u16ObjId = AuxFunction2_X1`
  - **Erklärung**: Liefert den booleschen Zustand des zugewiesenen physischen Aux-Tasters/Joysticks über die Ereignis-/Datenschnittstelle `IND`/`IN`.
- **DigitalOutput_Q1**: logiBUS-Digitalausgang (Typ: `logiBUS::io::DQ::logiBUS_QX`)
  - **Parameter**: `QI = TRUE`, `Output = Output_Q1`
  - **Erklärung**: Schaltet den physischen Ausgang Q1 entsprechend dem Aux-Zustand.

Sub-Bausteine: GreenWhiteBackground

- **GreenWhiteBackground** (Typ: `MyLib::sys::GreenWhiteBackground2`)
  - **Parameter**: `u16ObjIdA = AuxFunction2_X1`
  - **Erklärung**: Setzt die Hintergrundfarbe des Aux-Objekts abhängig vom booleschen Eingang `DI1` (grün = EIN, weiß = AUS). Da `AuxFunction2_X1` ein Auxiliary-Function-Type-2-Objekt ist, muss die Zuweisung sowohl auf dem VT-Bildschirm als auch am Aux-Handle (Joystick) selbst angezeigt werden – beide Anzeigen hängen an getrennten Verbindungen und werden daher intern über zwei Kommandos gesetzt (`Q_BackgroundColour` für das VT, `Q_BackgroundColourAux` für das Aux-Handle). Deshalb muss hier `GreenWhiteBackground2` (nicht `GreenWhiteBackground1`) verwendet werden.

Programmablauf und Verbindungen

1. `AuxFunction2_X1.IND` (Ereignis bei Zustandsänderung des Aux-Tasters) ist über zwei parallele Event-Connections direkt mit `DigitalOutput_Q1.REQ` **und** `GreenWhiteBackground.REQ` verbunden.
2. Der zugehörige Datenwert `AuxFunction2_X1.IN` wird ebenfalls parallel auf `DigitalOutput_Q1.OUT` und `GreenWhiteBackground.DI1` geführt.
3. Da klassische Event-/DataConnections – anders als AX-Adapter-Plugs – mehrere Ziele aus einer Quelle erlauben, ist hierfür kein expliziter Split-Baustein nötig (im Unterschied zur AX-Adapter-Variante `Uebung_010f_AX`, die dafür `AX_SPLIT_2` benötigt).

4. `GreenWhiteBackground` färbt daraufhin sowohl das VT-Objekt als auch das Aux-Handle grün bzw. weiß, je nach aktuellem Zustand.

Nachteil: `AuxFunction2_X1` muss zweimal angegeben werden (`Aux_IX.u16ObjId` und `GreenWhiteBackground.u16ObjIdA`).

Zusammenfassung

Die Übung zeigt, wie ein physischer Aux-Taster gleichzeitig einen digitalen Ausgang schaltet und über eine passende Hintergrundfarben-Rückmeldung auf VT-Bildschirm und Aux-Handle sichtbar gemacht wird. Im Gegensatz zur Adapter-Variante `Uebung_010f_AX` kommt die klassische Verdrahtung ohne Split-Baustein aus, da normale Event-/DataConnections beliebig viele Ziele erlauben. Wichtig bleibt die Wahl von `GreenWhiteBackground2` statt `GreenWhiteBackground1`, da Auxiliary-Function-Type-2-Objekte zwei getrennte Farbrückmeldungen benötigen.

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

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

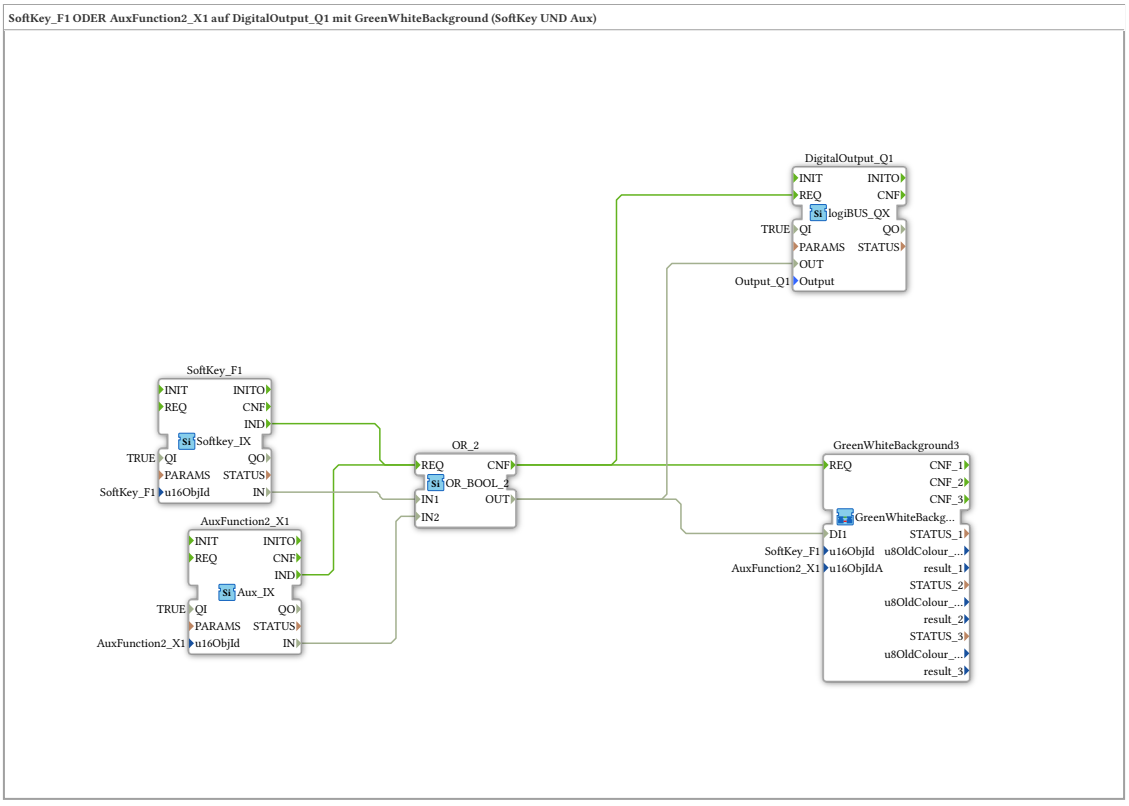
2.2.1.135 Uebung_010f2

Hinweis

Diese Übung existiert in den aktuellen Trainingsquellen nicht mehr (entfernt, zusammengelegt oder umbenannt). Diese Seite bleibt nur als Platzhalter für bestehende Verweise erhalten. Die aktuellen Übungen stehen in der Übungsübersicht.

2.2.1.136 Uebung_010f3: SoftKey_F1 ODER AuxFunction2_X1 auf DigitalOutput_Q1 mit GreenWhiteBackground (SoftKey UND Aux)

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_010f3 (SoftKey_F1 ODER AuxFunction2_X1 auf DigitalOutput_Q1 mit GreenWhiteBackground (SoftKey UND Aux)).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **SoftKey_F1 ODER AuxFunction2_X1 auf DigitalOutput_Q1 mit GreenWhiteBackground (SoftKey UND Aux)**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_010f3.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QX.
  - Parameter QI = TRUE
  - Parameter PARAMS =
  - Parameter Output = Output_Q1
- **SoftKey_F1**: Instanz des Typs isobus::UT::io::Softkey::Softkey_IX.
  - Parameter QI = TRUE
  - Parameter u16ObjId = SoftKey_F1
- **AuxFunction2_X1**: Instanz des Typs isobus::UT::io::Auxiliary::IN::Aux_IX.
  - Parameter QI = TRUE
  - Parameter u16ObjId = AuxFunction2_X1
- **OR_2**: Instanz des Typs iec61131::booleanOperators::OR_BOOL_2.

Verbindungen und Schnittstellen

Ereignisverbindungen:

- SoftKey_F1.IND -> OR_2.REQ
- AuxFunction2_X1.IND -> OR_2.REQ
- OR_2.CNF -> DigitalOutput_Q1.REQ
- OR_2.CNF -> GreenWhiteBackground3.REQ

Datenverbindungen:

- SoftKey_F1.IN -> OR_2.IN1
- AuxFunction2_X1.IN -> OR_2.IN2
- OR_2.OUT -> DigitalOutput_Q1.OUT
- OR_2.OUT -> GreenWhiteBackground3.DI1

Hinweise aus dem Modell

> SoftKey_F1 (VT) und AuxFunction2_X1 (Joystick) steuern denselben Q1 - ODER-verknuepft ueber OR_2. GreenWhiteBackground3 (nicht 1/2 einzeln!) bedient beide Hintergruende aus einem Baustein: u16ObjId=SoftKey (VT) und u16ObjIdA=Aux (VT+Aux-Handle).

Programmablauf und Funktionsweise

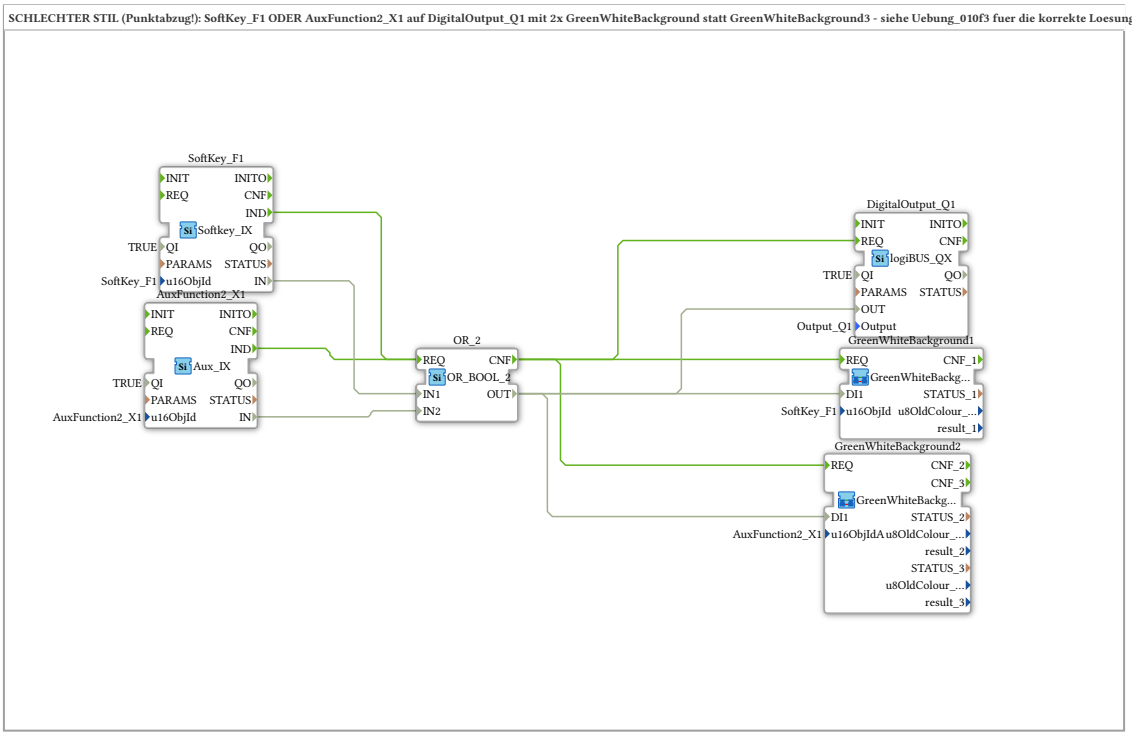
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_010f3 demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.2.1.137 Uebung_010f4: SCHLECHTER STIL (Punktabzug!): SoftKey_F1 ODER AuxFunction2_X1 auf DigitalOutput_Q1 mit 2x GreenWhiteBackground statt GreenWhiteBackground3 - siehe Uebung_010f3 fuer die korrekte Loesung

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_010f4 (SCHLECHTER STIL (Punktabzug!): SoftKey_F1 ODER AuxFunction2_X1 auf DigitalOutput_Q1 mit 2x GreenWhiteBackground statt GreenWhiteBackground3 - siehe Uebung_010f3 fuer die korrekte Loesung).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **SCHLECHTER STIL (Punktabzug!): SoftKey_F1 ODER AuxFunction2_X1 auf DigitalOutput_Q1 mit 2x GreenWhiteBackground statt GreenWhiteBackground3 - siehe Uebung_010f3 fuer die korrekte Loesung**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_010f4.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QX.
  - Parameter QI = TRUE
  - Parameter PARAMS =
  - Parameter Output = Output_Q1
- **SoftKey_F1**: Instanz des Typs isobus::UT::io::Softkey::Softkey_IX.
  - Parameter QI = TRUE
  - Parameter u16ObjId = SoftKey_F1
- **AuxFunction2_X1**: Instanz des Typs isobus::UT::io::Auxiliary::IN::Aux_IX.
  - Parameter QI = TRUE
  - Parameter u16ObjId = AuxFunction2_X1
- **OR_2**: Instanz des Typs iec61131::booleanOperators::OR_BOOL_2.

Verbindungen und Schnittstellen

Ereignisverbindungen:

- SoftKey_F1.IND -> OR_2.REQ
- AuxFunction2_X1.IND -> OR_2.REQ

- OR_2.CNF -> DigitalOutput_Q1.REQ
- OR_2.CNF -> GreenWhiteBackground1.REQ
- OR_2.CNF -> GreenWhiteBackground2.REQ

Datenverbindungen:

- SoftKey_F1.IN -> OR_2.IN1
- AuxFunction2_X1.IN -> OR_2.IN2
- OR_2.OUT -> DigitalOutput_Q1.OUT
- OR_2.OUT -> GreenWhiteBackground1.DI1
- OR_2.OUT -> GreenWhiteBackground2.DI1

Hinweise aus dem Modell

> SCHLECHTER STIL - Punktabzug! Hier wurden GreenWhiteBackground1 UND GreenWhiteBackground2 einzeln verdrahtet, obwohl es mit GreenWhiteBackground3 (ein Baustein, u16ObjId + u16ObjIdA) genau fuer diesen Fall (1 normales VT-Objekt + 1 Aux-Objekt) schon fertig existiert. Unnoetige Verdopplung der Farbauswahl-Logik. Korrekte Loesung: Uebung_010f3.

Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

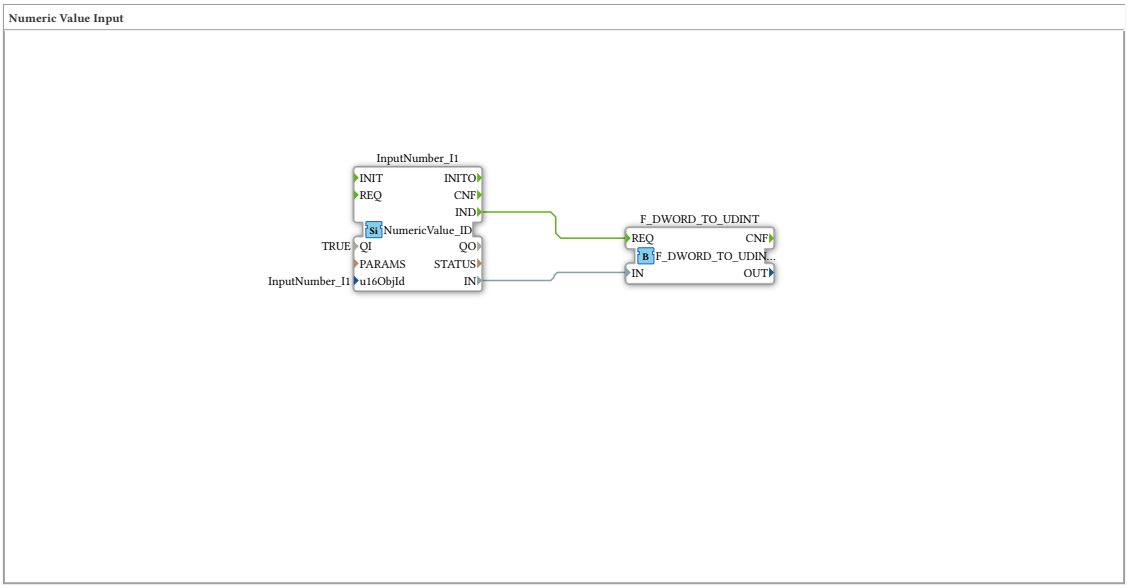
Die Übung Uebung_010f4 demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.2.1.138 Uebung_011: Numeric Value Input

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_011`. Hier wird demonstriert, wie Zahlenwerte (Daten) von einem ISOBUS-Terminal eingelesen werden.

Podcast



- ISO 11783-6: Softkeys und das Virtual Terminal verstehen – Dein Schlüssel zur Landmaschinen-Mechatronik
- Die drei Timer der DIN EN 61131-3 entschlüsselt – TP, TON & TOF präzise erklärt
- DIN EN 61131-3 vs. 61499-1: Dein Wegweiser durch die Normen der Industrieautomatisierung
- DIN EN 61131-3: Das Herz der Land- und Baumaschinen-Mechatronik und der Sprung in die Zukunft mit Ob
- FB_TOF und E_TOF: Verzögerungstimer in IEC 61131-3 und 61499

Ziel der Übung

Erlernen der Verarbeitung von numerischen Variablen im ISOBUS-Kontext. Es wird gezeigt, wie ein Nutzer am Terminal eine Zahl eingeben kann und wie diese Information als Daten-Ereignis-Kombination in der Steuerung ankommt.

Beschreibung und Komponenten

Die Subapplikation `Uebung_011.SUB` nutzt einen Eingabe-Baustein für numerische Werte.

Funktionsbausteine (FBs)

- **`InputNumber_I1`**: Typ `NumericValue_ID`. Dieser Baustein repräsentiert ein numerisches Eingabefeld (Data Mask Object) auf dem ISOBUS-Terminal. Sobald der Nutzer die Eingabe bestätigt, sendet der Baustein den neuen Wert am Port `IN` (DWORD) und feuert ein `IND`-Ereignis.
- **`F_DWORD_TO_UDINT`**: Ein Konvertierungs-Baustein, der den rohen 32-Bit-Wert vom Terminal in einen vorzeichenlosen Ganzzahlwert (UDINT) für die weitere Logik umwandelt.

Funktionsweise

Die Logik wartet auf die Bestätigung der Eingabe am Terminal:

```
<EventConnections>
    <Connection Source="InputNumber_I1.IND" Destination="F_DWORD_TO_UDINT.REQ"/>
</EventConnections>
<DataConnections>
    <Connection Source="InputNumber_I1.IN" Destination="F_DWORD_TO_UDINT.IN"/>
</DataConnections>
```

1. Der Nutzer tippt am Terminal auf das Zahlenfeld `I1`, gibt z.B. „42“ ein und drückt „Enter“.
2. Das Terminal sendet den Wert über den CAN-Bus an die Steuerung.
3. Der Baustein `InputNumber_I1` empfängt den Wert und löst das Ereignis `IND` aus.
4. Der Konvertierungs-Baustein übernimmt den Wert und stellt ihn der restlichen Applikation als Standard-Datentyp zur Verfügung.

Anwendungsbeispiel

Einstellung von Sollwerten: Der Landwirt gibt am Terminal die gewünschte Ausbringmenge für Saatgut (in kg/ha) oder die Zieltemperatur für die Getreidetrocknung ein. Die Software verarbeitet diesen numerischen Wert sofort weiter.

2.2.1.139 Uebung_011a: Numeric Value Output und BUTTON_PRESS_REPEAT_DONE



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_011a`. Hier wird die Interaktion zwischen Taster-Ereignissen und numerischen Anzeigen auf dem Terminal vertieft.

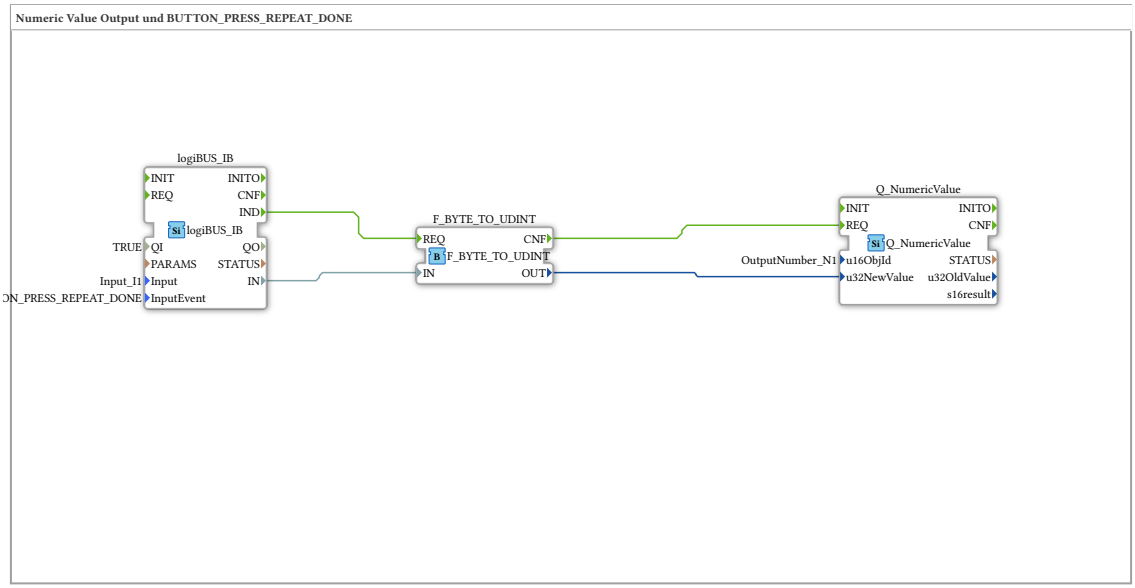
Ziel der Übung

Nutzung des `BUTTON_PRESS_REPEAT_DONE` Ereignisses zur Aktualisierung eines Anzeige-Objekts.

Beschreibung und Komponenten

In `Uebung_011a.SUB` wird ein Byte-Wert von einem Taster eingelesen und an eine numerische Anzeige am Terminal gesendet.

Funktionsbausteine (FBs)



- **`logiBUS_IB`**: Eingangsbaustein für Byte-Werte. Er ist auf das Event `BUTTON_PRESS_REPEAT_DONE` konfiguriert.
- **`Q_NumericValue`**: Ausgangsbaustein zur Anzeige einer Zahl auf dem Terminal.

Funktionsweise

Das Besondere ist die Wahl des Eingangs-Ereignisses:

- **`BUTTON_PRESS_REPEAT`**: Würde während des Drückens ständig Ereignisse senden (Blinker-Effekt).
- **`BUTTON_PRESS_REPEAT_DONE`**: Feuert nur **ein einziges Mal**, nämlich dann, wenn der Nutzer den Taster nach einer (eventuell wiederholten) Betätigung endgültig loslässt.

Die Logik sorgt dafür, dass der aktuelle Byte-Wert des Tasters (z.B. eine ID oder ein Zählerstand) erst am Ende der Interaktion an das Terminal übertragen wird.

Anwendungsbeispiel

Zähler-Übertragung: Ein Bediener hält einen Taster gedrückt, um einen Wert intern hochzuzählen. Damit der CAN-Bus nicht durch ständige Display-Updates belastet wird, erfolgt die Aktualisierung der Anzeige auf dem Terminal erst dann, wenn der Finger weggenommen wird (`REPEAT_DONE`).

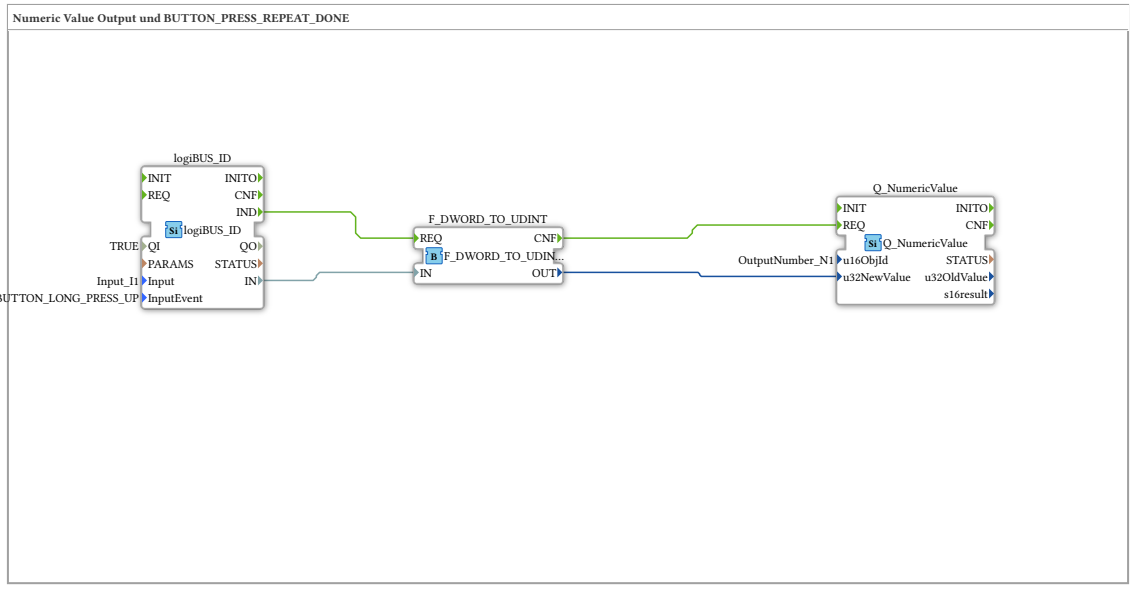
2.2.1.140 Uebung_011a2: Numeric Value Output und BUTTON_PRESS_REPEAT_DONE

NotebookLM

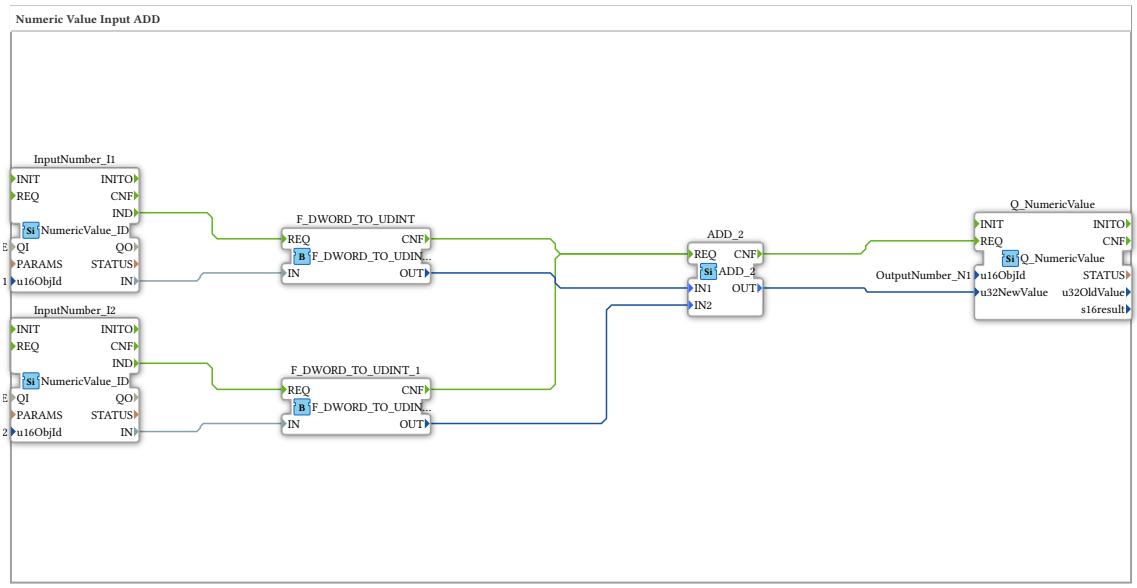
Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_011a2`.

Übersicht

Strukturell identisch mit `Uebung_011a`, verwendet diese Variante das Ereignis `BUTTON_LONG_PRESS_UP` am Eingangsbaustein `logiBUS_ID`. Die Anzeige auf dem Terminal wird hier erst dann aktualisiert, wenn ein zuvor erkannter langer Tastendruck durch Loslassen beendet wurde. Dies demonstriert eine weitere Variante der Interaktions-Bestätigung für numerische Werte.



2.2.1.141 Uebung_011b1: Numeric Value Input ADD



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert die Verarbeitung zweier numerischer Eingabewerte über den ISOBUS (UT). Die Werte werden als `DWORD` empfangen, in `UDINT` konvertiert, addiert und das Ergebnis als numerischer Ausgabewert bereitgestellt. Sie dient als einführendes Beispiel für die Kombination von Datentypkonvertierung, arithmetischen Operationen und der Nutzung der ISOBUS-NumericValue-Schnittstelle.

Verwendete Funktionsbausteine (FBs)

- **InputNumber_I1 / InputNumber_I2**
  - **Typ**: `isobus::UT::io::NumericValue::NumericValue_ID`
  - **Parameter**:
    - `QI` = `TRUE` (Eingang aktiviert)
    - `u16ObjId` = `"InputNumber_I1"` bzw. `"InputNumber_I2"` (jeweilige Objekt-ID)
  - **Funktion**: Stellt einen numerischen Eingabewert über den ISOBUS bereit. Bei einem eingehenden Ereignis (IND) wird der aktuelle Wert am Datenausgang `IN` (vom Typ `DWORD`) ausgegeben.

- **F_DWORD_TO_UDINT / F_DWORD_TO_UDINT_1**
  - **Typ**: `iec61131::conversion::F_DWORD_TO_UDINT`
  - **Parameter**: keine
  - **Funktion**: Konvertiert einen `DWORD`-Wert in einen `UDINT`-Wert. Der konvertierte Wert wird am Ausgang `OUT` ausgegeben. Die Konvertierung wird durch ein Ereignis am Eingang `REQ` gestartet; nach Abschluss wird der Ausgang `CNF` aktiviert.
- **ADD_2**
  - **Typ**: `iec61131::arithmetic::ADD_2`
  - **Parameter**: keine
  - **Funktion**: Addiert zwei `UDINT`-Werte an den Eingängen `IN1` und `IN2`. Das Ergebnis wird am Ausgang `OUT` (ebenfalls `UDINT`) ausgegeben. Ein Ereignis an `REQ` startet die Berechnung; nach Fertigstellung wird `CNF` aktiviert.
- **Q_NumericValue**
  - **Typ**: `isobus::UT::Q::Q_NumericValue`
  - **Parameter**:
    - `u16ObjId` = `"OutputNumber_N1"`
  - **Funktion**: Sendet einen numerischen Wert über den ISOBUS. Der zu sendende Wert wird am Daten-Eingang `u32NewValue` (vom Typ `UDINT`) erwartet. Ein Ereignis an `REQ` löst die Ausgabe aus; der Ausgang `CNF` bestätigt die erfolgreiche Übertragung.

Programmablauf und Verbindungen

Der Ablauf wird durch die Ereignis- und Datenverbindungen im Netzwerk gesteuert:

1. **Eingabe der Werte** – Die Funktionsbausteine `InputNumber_I1` und `InputNumber_I2` warten auf eingehende ISOBUS-Nachrichten. Sobald ein Wert anliegt, wird das Ereignis `IND` ausgelöst.
2. **Konvertierung** – Das Ereignis `IND` von `InputNumber_I1` triggert `F_DWORD_TO_UDINT` (über `REQ`). Gleichzeitig wird `F_DWORD_TO_UDINT_1` durch das `IND` von `InputNumber_I2` getriggert. Die konvertierten `UDINT`-Werte stehen an den Ausgängen `OUT` der Konverter an.
3. **Addition** – Nach Abschluss der Konvertierung (jeweiliges `CNF`-Ereignis) wird der Funktionsbaustein `ADD_2` über seinen Eingang `REQ` aufgerufen. Die konvertierten Werte der beiden Konverter werden mit den Datenverbindungen an `IN1` und `IN2` von `ADD_2` übergeben.
4. **Ausgabe** – Das `CNF`-Ereignis von `ADD_2` triggert den Baustein `Q_NumericValue`. An dessen Dateneingang `u32NewValue` liegt das Additionsergebnis an. Der Baustein sendet diesen Wert über den ISOBUS an die Objekt-ID `OutputNumber_N1`.

Hinweise für den Nutzer:

- Die Objekt-IDs (`InputNumber_I1`, `InputNumber_I2`, `OutputNumber_N1`) müssen mit den im ISOBUS-System konfigurierten Objekten übereinstimmen.
- Die Übung setzt Grundkenntnisse in der 4diac-IDE und der IEC 61499-Ereignissteuerung voraus.
- Schwierigkeitsgrad: Einsteiger.

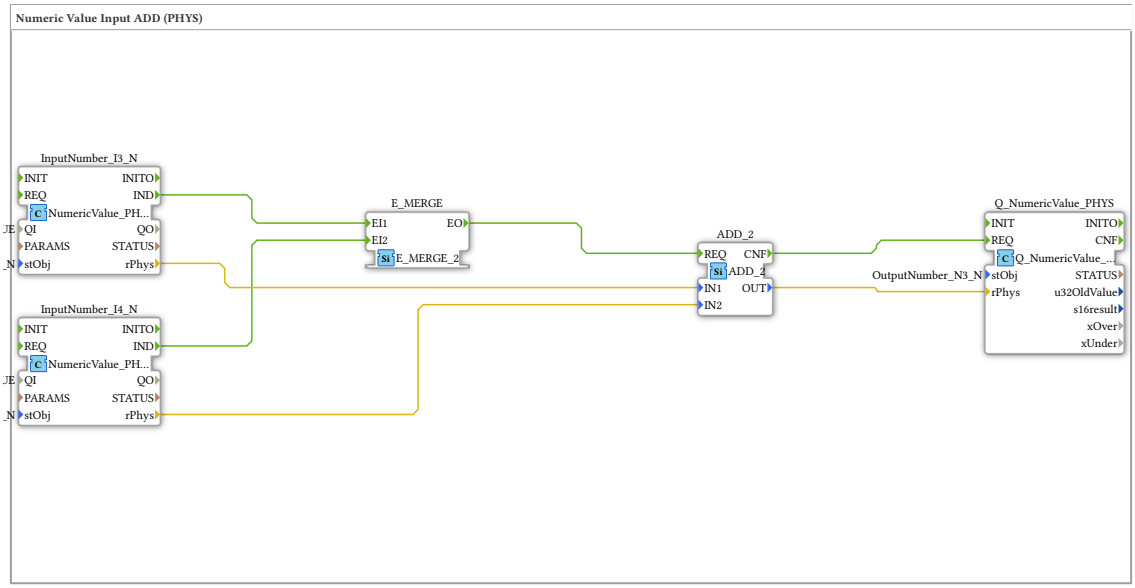
Zusammenfassung

Die Übung **Uebung_011b1** veranschaulicht den gesamten Datenpfad von der ISOBUS-Eingabe über Datentypkonvertierung und arithmetische Verarbeitung bis zur ISOBUS-Ausgabe. Sie ist ein typisches Beispiel für die strukturierte, ereignisgesteuerte Programmierung mit 4diac und IEC 61499. Die klare Trennung von Ereignis- und Datenflüssen erleichtert das Verständnis und die Wiederverwendbarkeit der Bausteine.

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

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

2.2.1.142 Uebung_011b1_PHYS: Numeric Value Input ADD (PHYS)



• – ◦ • – ◦ • – ◦ •

Einleitung

In dieser Übung wird eine Addition von zwei physikalischen Werten (REAL) mithilfe des NumericValue-Patterns aus der isobus-Bibliothek realisiert. Die Werte werden über die Bausteine `InputNumber_I3` und `InputNumber_I4` eingelesen, addiert und das Ergebnis über den Baustein `Q_NumericValue_PHYS` ausgegeben.

Verwendete Funktionsbausteine (FBs)

- **InputNumber_I3** (Typ: `isobus::UT::io::NumericValue::NumericValue_PHYS`)
  - Parameter:
    - `QI` = `TRUE`
    - `stObj` = `InputNumber_I3`
  - Ereignisausgang: `IND` (Indication) – wird bei einer Wertänderung ausgelöst
  - Datenausgang: `rPhys` (REAL) – der aktuelle physikalische Wert
- **InputNumber_I4** (Typ: `isobus::UT::io::NumericValue::NumericValue_PHYS`)
  - Parameter:
    - `QI` = `TRUE`
    - `stObj` = `InputNumber_I4`
  - Ereignisausgang: `IND`
  - Datenausgang: `rPhys`
- **ADD_2** (Typ: `iec61131::arithmetic::ADD_2`)
  - Parameter: keine
  - Ereigniseingang: `REQ` (Request) – löst die Addition aus
  - Ereignisausgang: `CNF` (Confirmation) – signalisiert die fertige Berechnung
  - Dateneingänge: `IN1` (REAL), `IN2` (REAL)
  - Datenausgang: `OUT` (REAL) – Summe der beiden Eingangswerte
- **Q_NumericValue_PHYS** (Typ: `isobus::UT::Q::Q_NumericValue_PHYS`)
  - Parameter:
    - `stObj` = `OutputNumber_N3`
  - Ereigniseingang: `REQ` – übernimmt den Wert bei Ereignis
  - Dateneingang: `rPhys` (REAL) – der zu setzende physikalische Wert

Funktionsweise

Die beiden Eingangsbausteine liefern bei einer Wertänderung ein Ereignis (`IND`). Dieses Ereignis wird an den Addierer `ADD_2` gesendet (`REQ`). Gleichzeitig werden die aktuellen physikalischen Werte (`rPhys`) an die Dateneingänge `IN1` und `IN2` von `ADD_2` übergeben. Nach der Berechnung sendet `ADD_2` eine Bestätigung (`CNF`) an den Ausgangsbaustein `Q_NumericValue_PHYS`, der das Ergebnis übernimmt und intern speichert.

Programmablauf und Verbindungen

Der Ablauf ist ereignisgesteuert:

1. Wenn sich der Wert an `InputNumber_I3` oder `InputNumber_I4` ändert, wird das `IND`-Ereignis ausgelöst.
2. Beide `IND`-Ereignisse sind mit dem `REQ`-Eingang von `ADD_2` verbunden (ODER-Verknüpfung – bereits eines löst die Addition aus).
3. `ADD_2` addiert die beiden REAL-Werte und gibt das Ergebnis an `OUT` aus.
4. Nach der Addition sendet `ADD_2` das `CNF`-Ereignis, welches den Baustein `Q_NumericValue_PHYS` triggert, den Ausgangswert zu setzen.

Datenverbindungen:

- `InputNumber_I3.rPhys` ⟶ `ADD_2.IN1`
- `InputNumber_I4.rPhys` ⟶ `ADD_2.IN2`
- `ADD_2.OUT` ⟶ `Q_NumericValue_PHYS.rPhys`

Ereignisverbindungen:

- `InputNumber_I3.IND` ⟶ `ADD_2.REQ`
- `InputNumber_I4.IND` ⟶ `ADD_2.REQ`
- `ADD_2.CNF` ⟶ `Q_NumericValue_PHYS.REQ`

Zusammenfassung

Die Übung demonstriert die Verwendung von physikalischen Werten (REAL) mit dem NumericValue-Pattern aus der isobus-Bibliothek. Zwei Eingänge werden addiert und das Ergebnis an einen Ausgang weitergegeben. Die PHYS-Variante arbeitet mit Gleitkommazahlen und ermöglicht so die Verarbeitung von kontinuierlichen Messwerten. Die ereignisgesteuerte Ausführung stellt sicher, dass die Berechnung nur bei Änderungen durchgeführt wird.

2.2.1.143 Uebung_011b2: Numeric Value Input MUL

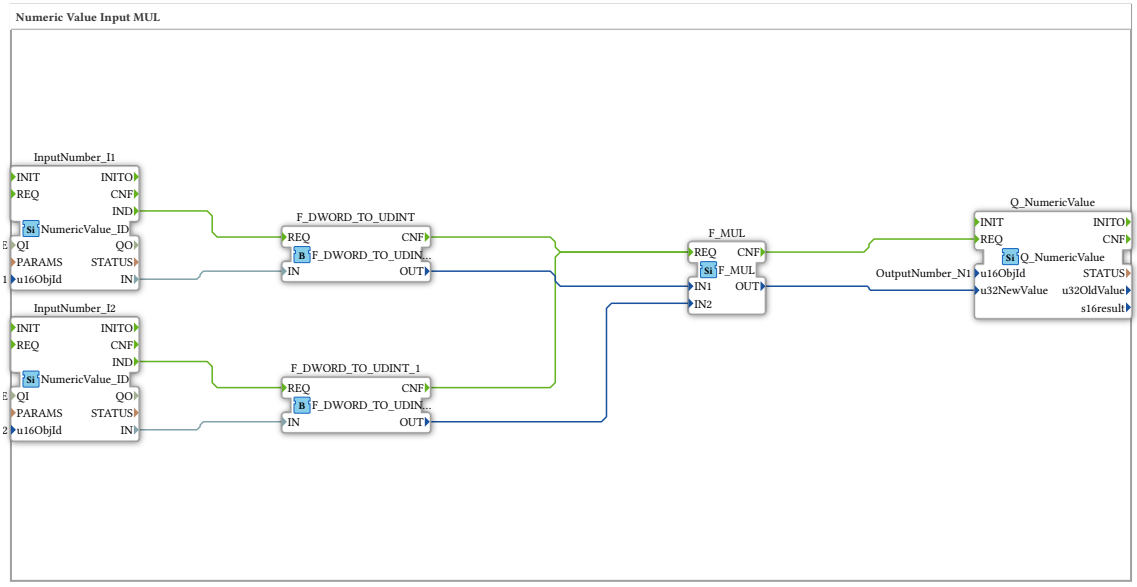


Bild folgt (sofern vorhanden)

• – ◦ • – ◦ • – ◦ •

Einleitung

Die Übung **Uebung_011b2** realisiert eine einfache Multiplikation zweier numerischer Werte. Zwei Eingänge (InputNumber_I1 und InputNumber_I2) lesen je einen DWORD-Wert aus dem ISOBUS-Netzwerk, wandeln diesen in den Datentyp UDINT um und multiplizieren die Ergebnisse miteinander. Das Produkt wird über einen Ausgang (OutputNumber_N1) wieder auf den Bus geschrieben. Die Übung zeigt die Verwendung von Ein-/Ausgabe-FB für numerische Werte und die arithmetische Verknüpfung über IEC 61131-Funktionsbausteine.

Verwendete Funktionsbausteine (FBs)

- **InputNumber_I1** (Typ: `isobus::UT::io::NumericValue::NumericValue_ID`)
  - Parameter: `QI` = `TRUE`, `u16ObjId` = `InputNumber_I1`
  - Ereignisausgang: `IND`
  - Datenausgang: `IN` (DWORD)
  - Funktion: Liest den aktuellen numerischen Wert des ISOBUS-Objekts „InputNumber_I1“ und stellt ihn als DWORD bereit.
- **InputNumber_I2** (Typ: `isobus::UT::io::NumericValue::NumericValue_ID`)
  - Parameter: `QI` = `TRUE`, `u16ObjId` = `InputNumber_I2`
  - Ereignisausgang: `IND`
  - Datenausgang: `IN` (DWORD)
  - Funktion: Liest den aktuellen numerischen Wert des ISOBUS-Objekts „InputNumber_I2“ und stellt ihn als DWORD bereit.
- **F_DWORD_TO_UDINT** (Typ: `iec61131::conversion::F_DWORD_TO_UDINT`)
  - Ereigniseingang: `REQ`, Ereignisausgang: `CNF`
  - Dateneingang: `IN` (DWORD), Datenausgang: `OUT` (UDINT)
  - Funktion: Wandelt den eingehenden DWORD-Wert in einen vorzeichenlosen 32-Bit-Integer (UDINT) um.
- **F_DWORD_TO_UDINT_1** (Typ: `iec61131::conversion::F_DWORD_TO_UDINT`)
  - Gleiche Konfiguration und Funktion wie oben, dient der Umwandlung des zweiten Eingangswerts.
- **F_MUL** (Typ: `iec61131::arithmetic::F_MUL`)
  - Ereigniseingang: `REQ`, Ereignisausgang: `CNF`
  - Dateneingänge: `IN1`, `IN2` (beide UDINT), Datenausgang: `OUT` (UDINT)
  - Funktion: Multipliziert die beiden eingehenden UDINT-Werte und gibt das Produkt als UDINT aus.
- **Q_NumericValue** (Typ: `isobus::UT::Q::Q_NumericValue`)
  - Parameter: `u16ObjId` = `OutputNumber_N1`
  - Ereigniseingang: `REQ`
  - Dateneingang: `u32NewValue` (UDINT)
  - Funktion: Schreibt den übergebenen numerischen Wert auf das ISOBUS-Objekt „OutputNumber_N1“.

Programmablauf und Verbindungen

1. **Ereignissteuerung**:
   - Sobald `InputNumber_I1` einen neuen Wert liefert, feuert dessen Ereignisausgang `IND`. Dieses Ereignis wird zum `REQ`-Eingang von `F_DWORD_TO_UDINT` verbunden.
   - Gleichzeitig triggert `InputNumber_I2.IND` den zweiten Umwandler `F_DWORD_TO_UDINT_1`.
   - Nach Abschluss der jeweiligen Konvertierung feuern die `CNF`-Ausgänge beider Umwandler – beide verbunden mit dem `REQ`-Eingang von `F_MUL`. (Hinweis: Die beiden Ereignisse werden beim Verbinden implizit ODER-verknüpft, sodass jede neue Eingabe eine Neuberechnung auslöst.)
   - Nach der Multiplikation feuert `F_MUL.CNF` und triggert den Ausgangs-FB `Q_NumericValue`.
2. **Datenfluss**:
   - Die Datenausgänge `IN` der Eingabe-FBs werden direkt auf die Dateneingänge `IN` der jeweiligen Umwandler gelegt.
   - Die Ausgänge `OUT` der Umwandler (UDINT) gelangen an `F_MUL.IN1` (aus `I1`) bzw. `F_MUL.IN2` (aus `I2`).

- Das Produkt `F_MUL.OUT` wird auf den Eingang `u32NewValue` von `Q_NumericValue` geschrieben und dort auf den Bus ausgegeben.

Die gesamte Logik ist ereignisgesteuert: Sobald ein neuer Messwert an einem der Eingänge anliegt, wird die gesamte Kette durchlaufen und der Ausgang aktualisiert.

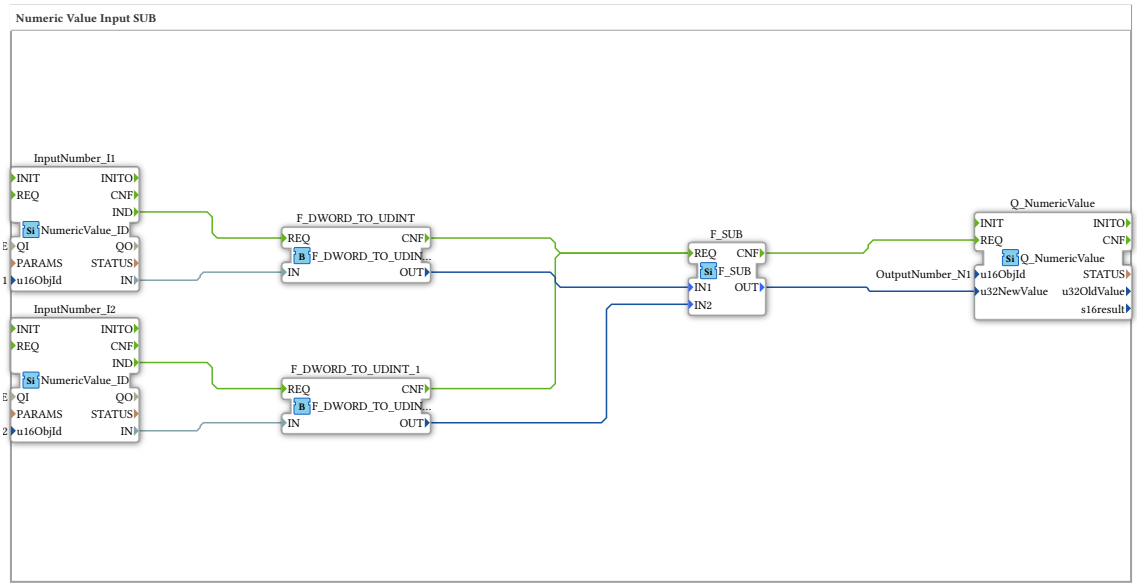
Zusammenfassung

Die Übung demonstriert den Umgang mit ISOBUS-NumericValue-FBs und IEC 61131-Konvertierungs- sowie Arithmetikbausteinen in einer 4diac-Subapplikation. Ziel ist die einfache Multiplikation zweier Buswerte. Durch die getrennte Ereignisverkettung wird sichergestellt, dass jeder neue Eingangswert sofort verarbeitet wird. Die Übung eignet sich als Grundlage für komplexere Berechnungen mit mehreren Eingängen und Ausgängen.

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

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

2.2.1.144 Übung_011b3: Numeric Value Input SUB



• – ○ • – ○ • – ○ •

Einleitung

Übung **Uebung_011b3** führt eine einfache Subtraktion zweier über das ISOBUS-Netzwerk eingelesener numerischer Werte durch. Anders als die vorherigen Übungen dieser Reihe dient sie dazu, einen echten Befund auf ESP32-P4-Hardware zu zeigen: `F_SUB` berechnet bei Unterlauf den in IEC 61131-3 spezifizierten Zweierkomplement-Wraparound, und für den hier verwendeten unsigned-Typ `UDINT` ist das gefährlich für reale Messwerte (Sollwertdifferenzen, Restweg-Berechnungen). Mit `InputNumber_I1 = 1` und `InputNumber_I2 = 12` gibt die Übung `UDINT#4294967285` aus statt des mathematisch erwarteten `-11` — weil `UDINT` keine negative Zahl darstellen kann und die normale Subtraktion überläuft, statt den Unterlauf zu melden. Dieser Befund hat direkt die SafeArithmetic-Bibliothek und ihren SAFE_SUB-Baustein motiviert — siehe Übung_011b6 für dasselbe Szenario, erneut ausgeführt mit `SAFE_SUB`.

Verwendete Funktionsbausteine (FBs)

- **InputNumber_I1** (Typ: `isobus::UT::io::NumericValue::NumericValue_ID`)
  - Parameter: `QI` = `TRUE`, `u16ObjId` = `InputNumber_I1`
  - Ereignis-Ausgang: `IND`, Daten-Ausgang: `IN` (DWORD)
  - Liest den aktuellen numerischen Wert des ISOBUS-Objekts „InputNumber_I1“.
- **InputNumber_I2** (Typ: `isobus::UT::io::NumericValue::NumericValue_ID`)
  - Wie oben, für das ISOBUS-Objekt „InputNumber_I2“.

- **F_DWORD_TO_UDINT / F_DWORD_TO_UDINT_1** (Typ: `iec61131::conversion::F_DWORD_TO_UDINT`)
  - Konvertieren die eingehenden DWORD-Werte nach UDINT.
- **F_SUB** (Typ: `iec61131::arithmetic::F_SUB`)
  - Daten-Eingänge: `IN1` (Minuend), `IN2` (Subtrahend), beide UDINT. Daten-Ausgang: `OUT` (UDINT).
  - Berechnet `IN1 - IN2` mit normaler IEC-61131-3-Semantik (Wraparound).
- **Q_NumericValue** (Typ: `isobus::UT::Q::Q_NumericValue`)
  - Parameter: `u16ObjId` = `OutputNumber_N1`. Schreibt `F_SUB.OUT` zurück auf das ISOBUS-Netzwerk.

Programmablauf und Verbindungen

1. **Ereignissteuerung**: `InputNumber_I1.IND`/`InputNumber_I2.IND` lösen jeweils das `REQ` ihres `F_DWORD_TO_UDINT`-Konverters aus. Beide `CNF`-Ausgänge der Konverter sind mit `F_SUB.REQ` verbunden (implizit ODER-verknüpft — jeder neue Eingang löst eine Neuberechnung aus). `F_SUB.CNF` löst `Q_NumericValue.REQ` aus.
2. **Datenfluss**: `InputNumber_I1.IN`/`InputNumber_I2.IN` (DWORD) gehen an die `IN`-Eingänge der Konverter. Die `OUT`-Werte (UDINT) der Konverter gehen an `F_SUB.IN1` (von I1) und `F_SUB.IN2` (von I2). `F_SUB.OUT` geht an `Q_NumericValue.u32NewValue`.

Hardware-Befund

`InputNumber_I1 = UDINT#1`, `InputNumber_I2 = UDINT#12`:
`F_SUB.OUT = UDINT#1 - UDINT#12 = UDINT#4294967285  (F_SUB hat kein LIMIT_HIT)`
Das ist der korrekte, in IEC 61131-3 spezifizierte Zweierkomplement-Wraparound für `UDINT` — kein FORTE-Bug — aber es erzeugt still ein unsinniges Ergebnis für eine reale Subtraktion, deren wahre Antwort negativ und in einem unsigned-Typ nicht darstellbar ist. Siehe Übung_011b6 für die Behebung.

Zusammenfassung

Diese Übung ist der Ausgangspunkt einer kleinen Reihe: sie reproduziert auf echter Hardware genau das Problem des stillen Wraparounds, das die SafeArithmetic-Bibliothek lösen soll. Vergleichen Sie ihre Ausgabe direkt mit Übung_011b6 (gleiche Eingänge, `SAFE_SUB` statt `F_SUB`).

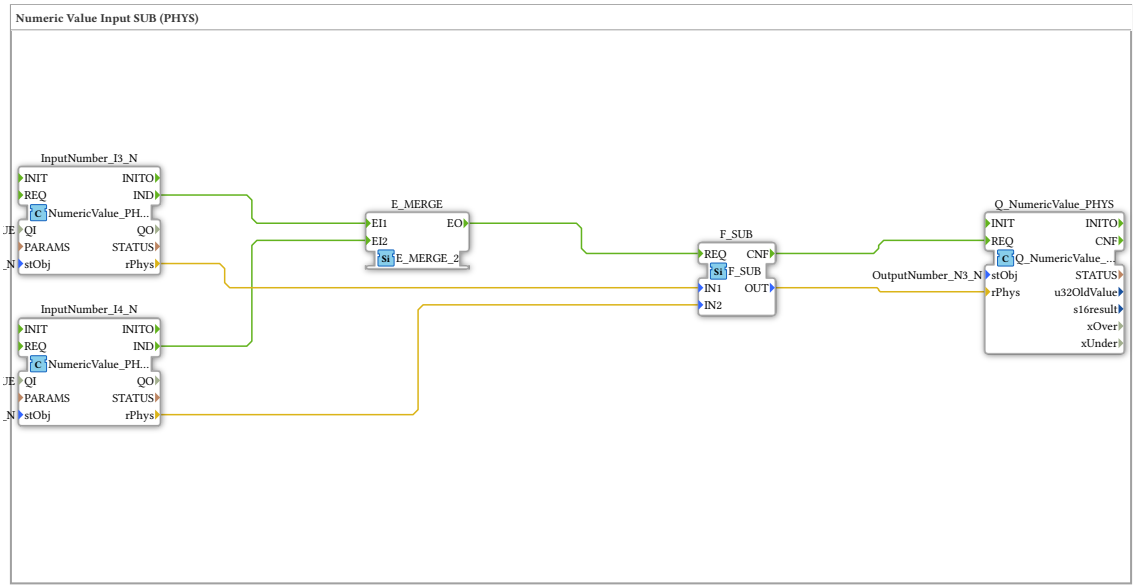
2.2.1.145 Uebung_011b3_PHYS: Numeric Value Input SUB (PHYS)

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_011b3_PHYS`. Zwei am VT-Terminal eingegebene physikalische Zahlenwerte werden voneinander subtrahiert und das Ergebnis wieder angezeigt.

Ziel der Übung

Zwei Zahlenfelder (`InputNumber_I3_N`, `InputNumber_I4_N`) am VT-Terminal entgegennehmen, die Differenz bilden und das Ergebnis in einem dritten Zahlenfeld anzeigen.

Beschreibung und Komponenten

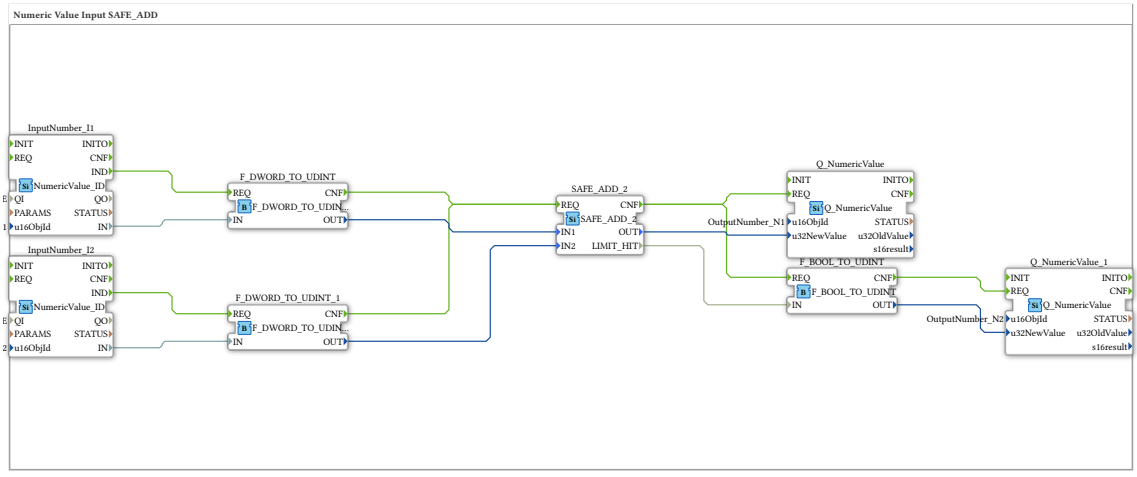


- **`InputNumber_I3_N`**, **`InputNumber_I4_N`**: `isobus::UT::io::NumericValue::NumericValue_PHYS`, physikalische Zahlenfelder auf dem VT.
- **`F_SUB`**: `iec61131::arithmetic::F_SUB`, subtrahiert die beiden eingegebenen Werte.
- **`E_MERGE`**: `iec61499::events::E_MERGE_2`, löst die Berechnung aus, sobald eines der beiden Felder geändert wurde.
- **`Q_NumericValue_PHYS`**: zeigt das Ergebnis der Subtraktion an (`OutputNumber_N3_N`).

Funktionsweise

1. Der Bediener ändert eines der beiden Zahlenfelder `InputNumber_I3_N`/`InputNumber_I4_N`.
2. `E_MERGE` löst daraufhin `F_SUB` aus, das die Differenz `IN1 - IN2` berechnet.
3. Das Ergebnis wird in `Q_NumericValue_PHYS` angezeigt.

2.2.1.146 Übung_011b4: Numeric Value Input SAFE_ADD



• — ○ • — ○ • — ○ •

Einleitung

Übung **Uebung_011b4** führt eine Addition zweier über das ISOBUS-Netzwerk eingelesener numerischer Werte durch, unter Verwendung von SAFE_ADD_2 statt des normalen `ADD_2`/`F_ADD`. Anders als die vorherigen Übungen dieser Reihe zeigt sie einen zweiten Ausgang: `SAFE_ADD_2.LIMIT_HIT` wird nach `UDINT` konvertiert und auf `OutputNumber_N2` geschrieben, sodass ein Overflow direkt auf dem ISOBUS-

Netzwerk sichtbar wird als `1` (geklemmt) oder `0` (normal) — nicht nur still in einer übergelaufenen Summe auf `OutputNumber_N1` verborgen.

Verwendete Funktionsbausteine (FBs)

- **InputNumber_I1** / **InputNumber_I2** (Typ: `isobus::UT::io::NumericValue::NumericValue_ID`)
  - Lesen die aktuellen numerischen Werte der ISOBUS-Objekte „InputNumber_I1“/„InputNumber_I2“.
- **F_DWORD_TO_UDINT** / **F_DWORD_TO_UDINT_1** (Typ: `iec61131::conversion::F_DWORD_TO_UDINT`)
  - Konvertieren die eingehenden DWORD-Werte nach UDINT.
- **SAFE_ADD_2** (Typ: `SafeArithmetic::arithmetic::SAFE_ADD_2`)
  - Daten-Eingänge: `IN1`, `IN2` (beide UDINT). Daten-Ausgänge: `OUT` (UDINT, geklemmte Summe), `LIMIT_HIT` (BOOL, TRUE wenn die Addition übergelaufen ist).
- **Q_NumericValue** (Typ: `isobus::UT::Q::Q_NumericValue`, `u16ObjId` = `OutputNumber_N1`)
  - Schreibt `SAFE_ADD_2.OUT` zurück auf das ISOBUS-Netzwerk.
- **F_BOOL_TO_UDINT** (Typ: `iec61131::conversion::F_BOOL_TO_UDINT`)
  - Konvertiert `SAFE_ADD_2.LIMIT_HIT` (BOOL) nach `UDINT` (`1`/`0`).
- **Q_NumericValue_1** (Typ: `isobus::UT::Q::Q_NumericValue`, `u16ObjId` = `OutputNumber_N2`)
  - Schreibt das konvertierte `LIMIT_HIT` zurück auf das ISOBUS-Netzwerk.

Programmablauf und Verbindungen

1. **Ereignissteuerung**: `InputNumber_I1.IND`/`InputNumber_I2.IND` lösen das `REQ` ihrer Konverter aus. Beide `CNF`-Ausgänge lösen `SAFE_ADD_2.REQ` aus. `SAFE_ADD_2.CNF` löst sowohl `Q_NumericValue.REQ` (die Summe) als auch `F_BOOL_TO_UDINT.REQ` (das Grenzwert-Flag) aus, und `F_BOOL_TO_UDINT.CNF` löst `Q_NumericValue_1.REQ` aus.
2. **Datenfluss**: die konvertierten `IN1`/`IN2` (UDINT) gehen an `SAFE_ADD_2.IN1`/`IN2`. `SAFE_ADD_2.OUT` geht an `Q_NumericValue.u32NewValue`. `SAFE_ADD_2.LIMIT_HIT` geht an `F_BOOL_TO_UDINT.IN`, dessen `OUT` an `Q_NumericValue_1.u32NewValue` geht.

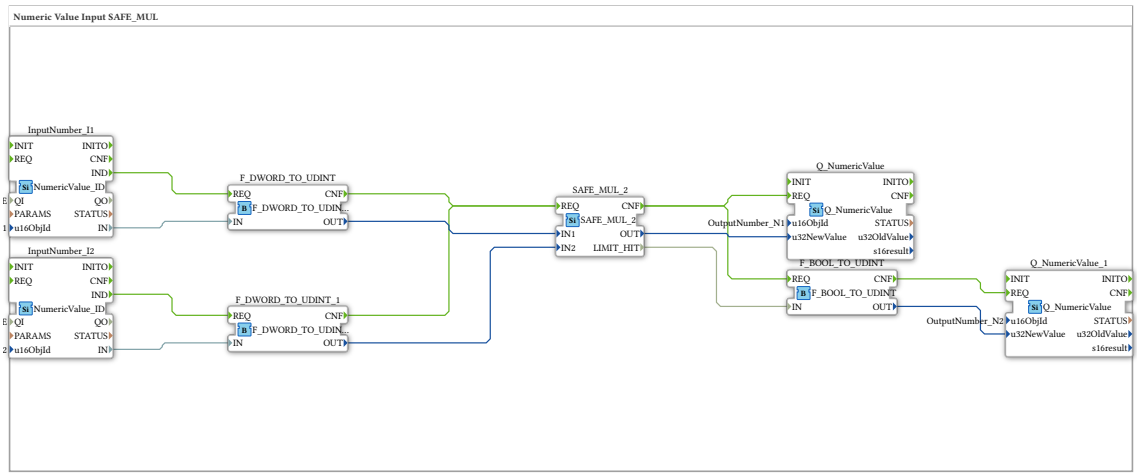
Erwartetes Verhalten

- Normalfall (z.B. `InputNumber_I1 = 100`, `InputNumber_I2 = 200`): `OutputNumber_N1 = 300`, `OutputNumber_N2 = 0`.
- Overflow-Fall (z.B. `InputNumber_I1 = UDINT#4294967290`, `InputNumber_I2 = 100`): die wahre Summe überschreitet den `UDINT`-Bereich, daher `OutputNumber_N1 = UDINT#4294967295` (auf max geklemmt) und `OutputNumber_N2 = 1` — der Overflow ist jetzt sichtbar, statt still auf eine kleine, plausibel aussehende (aber falsche) Zahl überzulaufen.

Zusammenfassung

Diese Übung zeigt `SAFE_ADD_2` als Drop-in-Ersatz für `ADD_2`/`F_ADD`, der aus einem stillen, schwer bemerkbaren Wraparound ein explizites, beobachtbares `LIMIT_HIT`-Signal macht.

2.2.1.147 Übung_011b5: Numeric Value Input SAFE_MUL



• – ○ • – ○ • – ○ •

Einleitung

Übung **Uebung_011b5** führt eine Multiplikation zweier über das ISOBUS-Netzwerk eingelesener numerischer Werte durch, unter Verwendung von SAFE_MUL_2 statt des normalen `F_MUL` (vergleiche Übung_011b2, die `F_MUL` für denselben Aufbau nutzt). `SAFE_MUL_2.LIMIT_HIT` wird nach `UDINT` konvertiert und auf `OutputNumber_N2` geschrieben, sodass ein Overflow direkt auf dem ISOBUS-Netzwerk sichtbar wird als `1` (geklemmt) oder `0` (normal).

Verwendete Funktionsbausteine (FBs)

- **InputNumber_I1** / **InputNumber_I2** (Typ: `isobus::UT::io::NumericValue::NumericValue_ID`)
  - Lesen die aktuellen numerischen Werte der ISOBUS-Objekte „InputNumber_I1“/„InputNumber_I2“.
- **F_DWORD_TO_UDINT** / **F_DWORD_TO_UDINT_1** (Typ: `iec61131::conversion::F_DWORD_TO_UDINT`)
  - Konvertieren die eingehenden DWORD-Werte nach UDINT.
- **SAFE_MUL_2** (Typ: `SafeArithmetic::arithmetic::SAFE_MUL_2`)
  - Daten-Eingänge: `IN1`, `IN2` (beide UDINT). Daten-Ausgänge: `OUT` (UDINT, geklemmtes Produkt), `LIMIT_HIT` (BOOL, TRUE wenn die Multiplikation übergelaufen ist).
- **Q_NumericValue** (Typ: `isobus::UT::Q::Q_NumericValue`, `u16ObjId` = `OutputNumber_N1`)
  - Schreibt `SAFE_MUL_2.OUT` zurück auf das ISOBUS-Netzwerk.
- **F_BOOL_TO_UDINT** (Typ: `iec61131::conversion::F_BOOL_TO_UDINT`)
  - Konvertiert `SAFE_MUL_2.LIMIT_HIT` (BOOL) nach `UDINT` (`1`/`0`).
- **Q_NumericValue_1** (Typ: `isobus::UT::Q::Q_NumericValue`, `u16ObjId` = `OutputNumber_N2`)
  - Schreibt das konvertierte `LIMIT_HIT` zurück auf das ISOBUS-Netzwerk.

Programmablauf und Verbindungen

1. **Ereignissteuerung**: `InputNumber_I1.IND`/`InputNumber_I2.IND` lösen das `REQ` ihrer Konverter aus. Beide `CNF`-Ausgänge lösen `SAFE_MUL_2.REQ` aus. `SAFE_MUL_2.CNF` löst sowohl `Q_NumericValue.REQ` (das Produkt) als auch `F_BOOL_TO_UDINT.REQ` (das Grenzwert-Flag) aus, und `F_BOOL_TO_UDINT.CNF` löst `Q_NumericValue_1.REQ` aus.
2. **Datenfluss**: die konvertierten `IN1`/`IN2` (UDINT) gehen an `SAFE_MUL_2.IN1`/`IN2`. `SAFE_MUL_2.OUT` geht an `Q_NumericValue.u32NewValue`. `SAFE_MUL_2.LIMIT_HIT` geht an `F_BOOL_TO_UDINT.IN`, dessen `OUT` an `Q_NumericValue_1.u32NewValue` geht.

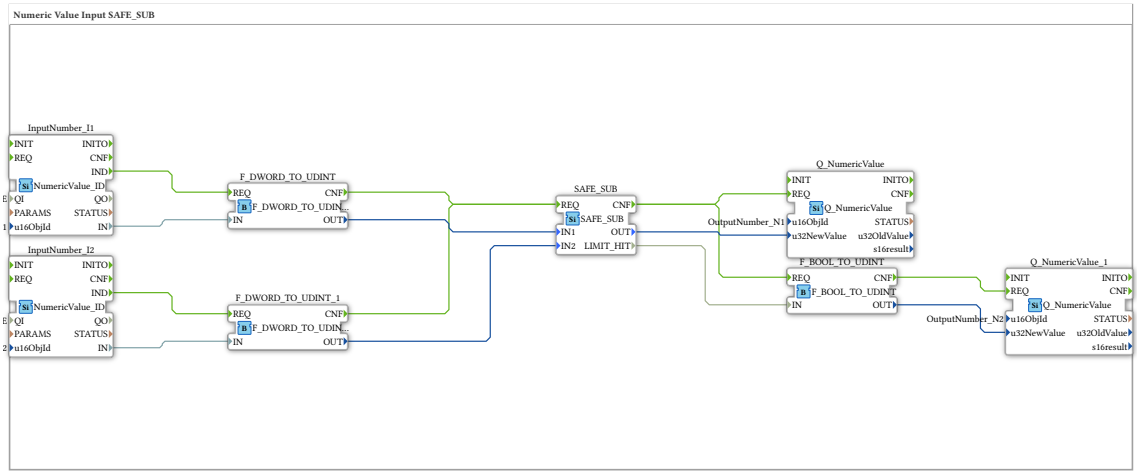
Erwartetes Verhalten

- Normalfall (z.B. `InputNumber_I1 = 100`, `InputNumber_I2 = 200`): `OutputNumber_N1 = 20000`, `OutputNumber_N2 = 0`.
- Overflow-Fall (z.B. `InputNumber_I1 = 100000`, `InputNumber_I2 = 100000`): das wahre Produkt (`10.000.000.000`) überschreitet den `UDINT`-Bereich, daher `OutputNumber_N1 = UDINT#4294967295` (auf max geklemmt) und `OutputNumber_N2 = 1`, statt des stillen `UDINT#1410065408` bei `F_MUL` (`10.000.000.000 mod 2^32`).

Zusammenfassung

Diese Übung zeigt `SAFE_MUL_2` als Drop-in-Ersatz für `F_MUL`, der aus einem stillen Wraparound (der bei Multiplikation ein Ergebnis erzeugen kann, das völlig unabhängig von beiden Eingängen aussieht) ein explizites, beobachtbares `LIMIT_HIT`-Signal macht.

2.2.1.148 Übung_011b6: Numeric Value Input SAFE_SUB



• – ○ • – ○ • – ○ •

Einleitung

Übung **Uebung_011b6** ist die direkte Behebung von Übung_011b3: dieselbe Subtraktion zweier über das ISOBUS-Netzwerk eingelesener numerischer Werte, aber mit SAFE_SUB statt normalem `F_SUB`. `SAFE_SUB.LIMIT_HIT` wird nach `UDINT` konvertiert und auf `OutputNumber_N2` geschrieben, sodass der Unterlauf, den `Uebung_011b3` still erzeugt hat, jetzt direkt auf dem ISOBUS-Netzwerk sichtbar ist.

Verwendete Funktionsbausteine (FBs)

- **InputNumber_I1** / **InputNumber_I2** (Typ: `isobus::UT::io::NumericValue::NumericValue_ID`)
  - Lesen die aktuellen numerischen Werte der ISOBUS-Objekte „InputNumber_I1“/„InputNumber_I2“.
- **F_DWORD_TO_UDINT** / **F_DWORD_TO_UDINT_1** (Typ: `iec61131::conversion::F_DWORD_TO_UDINT`)
  - Konvertieren die eingehenden DWORD-Werte nach UDINT.
- **SAFE_SUB** (Typ: `SafeArithmetic::arithmetic::SAFE_SUB`)
  - Daten-Eingänge: `IN1` (Minuend), `IN2` (Subtrahend), beide UDINT. Daten-Ausgänge: `OUT` (UDINT, geklemmte Differenz), `LIMIT_HIT` (BOOL, TRUE wenn die Subtraktion unter-/übergelaufen ist).
- **Q_NumericValue** (Typ: `isobus::UT::Q::Q_NumericValue`, `u16ObjId` = `OutputNumber_N1`)
  - Schreibt `SAFE_SUB.OUT` zurück auf das ISOBUS-Netzwerk.
- **F_BOOL_TO_UDINT** (Typ: `iec61131::conversion::F_BOOL_TO_UDINT`)
  - Konvertiert `SAFE_SUB.LIMIT_HIT` (BOOL) nach `UDINT` (`1`/`0`).
- **Q_NumericValue_1** (Typ: `isobus::UT::Q::Q_NumericValue`, `u16ObjId` = `OutputNumber_N2`)
  - Schreibt das konvertierte `LIMIT_HIT` zurück auf das ISOBUS-Netzwerk.

Programmablauf und Verbindungen

1. **Ereignissteuerung**: `InputNumber_I1.IND`/`InputNumber_I2.IND` lösen das `REQ` ihrer Konverter aus. Beide `CNF`-Ausgänge lösen `SAFE_SUB.REQ` aus. `SAFE_SUB.CNF` löst sowohl `Q_NumericValue.REQ` (die Differenz) als auch `F_BOOL_TO_UDINT.REQ` (das Grenzwert-Flag) aus, und `F_BOOL_TO_UDINT.CNF` löst `Q_NumericValue_1.REQ` aus.
2. **Datenfluss**: die konvertierten `IN1`/`IN2` (UDINT) gehen an `SAFE_SUB.IN1`/`IN2`. `SAFE_SUB.OUT` geht an `Q_NumericValue.u32NewValue`. `SAFE_SUB.LIMIT_HIT` geht an `F_BOOL_TO_UDINT.IN`, dessen `OUT` an `Q_NumericValue_1.u32NewValue` geht.

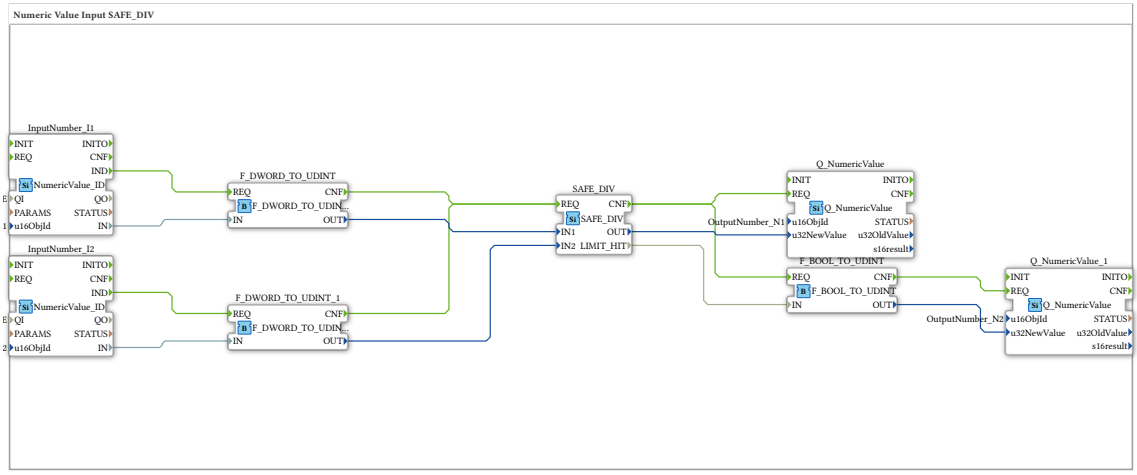
Hardware-Ergebnis (gleiche Eingänge wie Übung_011b3)

`InputNumber_I1 = UDINT#1`, `InputNumber_I2 = UDINT#12`:
`SAFE_SUB.OUT = 0` (geklemmt: unsigned-Subtraktion kann nur unterlaufen)
`SAFE_SUB.LIMIT_HIT = TRUE` -> `OutputNumber_N1 = 0`, `OutputNumber_N2 = 1`
Bestätigt direkt die Behebung: statt des stillen `UDINT#4294967285` aus Übung_011b3 wird das wahrhaft-negative und nicht darstellbare Ergebnis jetzt auf `0` geklemmt, mit einem expliziten, beobachtbaren `LIMIT_HIT = TRUE`.

Zusammenfassung

Führen Sie Übung_011b3 und diese Übung mit denselben Eingängen nebeneinander aus, um das exakte Vorher/Nachher der `SafeArithmetic`-Fix für den ursprünglichen Hardware-Befund zu sehen.

2.2.1.149 Übung_011b7: Numeric Value Input SAFE_DIV



• – ○ • – ○ • – ○ •

Einleitung

Übung **Uebung_011b7** führt eine Division zweier über das ISOBUS-Netzwerk eingelesener numerischer Werte durch, unter Verwendung von SAFE_DIV. Es gibt in dieser Reihe keine normale, nicht-sichere `F_DIV`-Vorgänger-Übung — Integer-Division durch Null ist in reinem C++ undefiniertes Verhalten und würde ein natives `F_DIV` zum Absturz bringen, daher existiert diese Übung ausschließlich, um das Division-durch-Null-Verhalten von `SAFE_DIV` direkt zu zeigen. `SAFE_DIV.LIMIT_HIT` wird nach `UDINT` konvertiert und auf `OutputNumber_N2` geschrieben.

Verwendete Funktionsbausteine (FBs)

- **InputNumber_I1** / **InputNumber_I2** (Typ: `isobus::UT::io::NumericValue::NumericValue_ID`)
  - Lesen die aktuellen numerischen Werte der ISOBUS-Objekte „InputNumber_I1“/„InputNumber_I2“.
- **F_DWORD_TO_UDINT** / **F_DWORD_TO_UDINT_1** (Typ: `iec61131::conversion::F_DWORD_TO_UDINT`)
  - Konvertieren die eingehenden DWORD-Werte nach UDINT.
- **SAFE_DIV** (Typ: `SafeArithmetic::arithmetic::SAFE_DIV`)
  - Daten-Eingänge: `IN1` (Dividend), `IN2` (Divisor), beide UDINT. Daten-Ausgänge: `OUT` (UDINT, Quotient, `0` wenn `IN2` gleich `0` ist), `LIMIT_HIT` (BOOL, TRUE bei Overflow oder Division durch Null).
- **Q_NumericValue** (Typ: `isobus::UT::Q::Q_NumericValue`, `u16ObjId` = `OutputNumber_N1`)
  - Schreibt `SAFE_DIV.OUT` zurück auf das ISOBUS-Netzwerk.
- **F_BOOL_TO_UDINT** (Typ: `iec61131::conversion::F_BOOL_TO_UDINT`)
  - Konvertiert `SAFE_DIV.LIMIT_HIT` (BOOL) nach `UDINT` (`1`/`0`).
- **Q_NumericValue_1** (Typ: `isobus::UT::Q::Q_NumericValue`, `u16ObjId` = `OutputNumber_N2`)
  - Schreibt das konvertierte `LIMIT_HIT` zurück auf das ISOBUS-Netzwerk.

Programmablauf und Verbindungen

1. **Ereignissteuerung**: `InputNumber_I1.IND`/`InputNumber_I2.IND` lösen das `REQ` ihrer Konverter aus. Beide `CNF`-Ausgänge lösen `SAFE_DIV.REQ` aus. `SAFE_DIV.CNF` löst sowohl `Q_NumericValue.REQ` (den Quotienten) als auch `F_BOOL_TO_UDINT.REQ` (das Grenzwert-Flag) aus, und `F_BOOL_TO_UDINT.CNF` löst `Q_NumericValue_1.REQ` aus.
2. **Datenfluss**: die konvertierten `IN1`/`IN2` (UDINT) gehen an `SAFE_DIV.IN1`/`IN2`. `SAFE_DIV.OUT` geht an `Q_NumericValue.u32NewValue`. `SAFE_DIV.LIMIT_HIT` geht an `F_BOOL_TO_UDINT.IN`, dessen `OUT` an `Q_NumericValue_1.u32NewValue` geht.

Hardware-Ergebnis

`InputNumber_I1 = UDINT#12`, `InputNumber_I2 = UDINT#0`:

```
SAFE_DIV.OUT = 0           (geklemmt: Division durch Null, statt Absturz)
SAFE_DIV.LIMIT_HIT = TRUE  ->  OutputNumber_N1 = 0, OutputNumber_N2 = 1
```

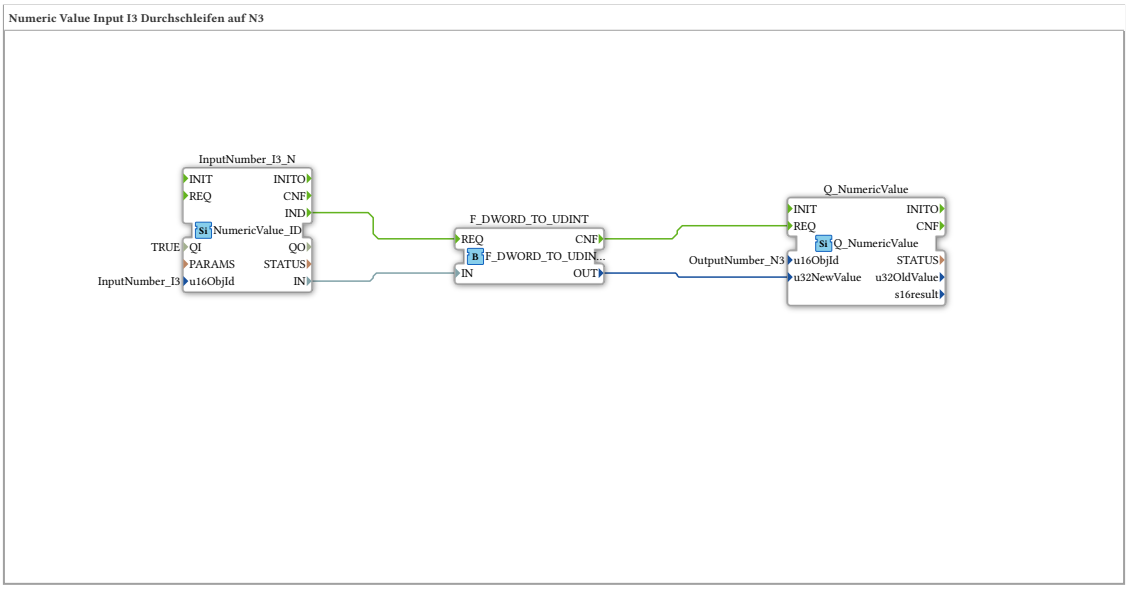
`InputNumber_I1 = UDINT#12`, `InputNumber_I2 = UDINT#4` (Normalfall):

```
SAFE_DIV.OUT = 3, SAFE_DIV.LIMIT_HIT = FALSE  ->  OutputNumber_N1 = 3, OutputNumber_N2 =
0
```

Zusammenfassung

Diese Übung zeigt, wie `SAFE_DIV` eine Division durch Null als explizite, beobachtbare Klemm-Bedingung behandelt, statt des undefinierten Verhaltens, das eine native Integer-Division durch Null sonst wäre.

2.2.1.150 Uebung_011c: Numeric Value Input I3 Durchschleifen auf N3



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung demonstriert das einfache Durchschleifen eines numerischen Werts von einem ISOBUS-Input-Objekt (InputNumber_I3) zu einem Output-Objekt (OutputNumber_N3). Dabei wird der vom Bus kommende DWORD-Wert in einen UDINT konvertiert, bevor er an das Ausgangsobjekt weitergegeben wird. Die Übung ist ein grundlegendes Beispiel für den Umgang mit dem **4diac-IDE** im Kontext von ISOBUS (ISO 11783) und zeigt, wie Funktionsbausteine zur Datenverarbeitung und -weiterleitung verschaltet werden.

Verwendete Funktionsbausteine (FBs)

InputNumber_I3

- **Typ**: `isobus::UT::io::NumericValue::NumericValue_ID`
- **Parameter**:
  - `QI` = `TRUE`
  - `u16ObjId` = `InputNumber_I3`
- **Ereigniseingänge**: (implizit durch den Typ, Standard: INIT, REQ, etc.)
- **Ereignisausgänge**: `IND` (wird ausgelöst, wenn ein neuer Wert vom Bus ankommt)
- **Dateneingänge**: (keine zusätzlichen außer den impliziten)
- **Datenausgänge**: `IN` (ausgehender Wert als DWORD)
- **Funktionsweise**: Dieser FB liest den aktuellen Wert des ISOBUS-Objekts mit der ID `InputNumber_I3`. Bei jeder Wertänderung am Bus wird das Ereignis `IND` ausgelöst und der aktuelle Wert als DWORD am Ausgang `IN` bereitgestellt.

F_DWORD_TO_UDINT

- **Typ**: `iec61131::conversion::F_DWORD_TO_UDINT`
- **Parameter**: keine zusätzlichen

- **Ereigniseingänge**: `REQ` (Konvertierung anfordern)
- **Ereignisausgänge**: `CNF` (Bestätigung nach erfolgreicher Konvertierung)
- **Dateneingänge**: `IN` (DWORD-Wert)
- **Datenausgänge**: `OUT` (konvertierter UDINT-Wert)
- **Funktionsweise**: Der FB wandelt den 32-Bit DWORD in einen vorzeichenlosen 32-Bit Integer (UDINT) um. Die Konvertierung wird durch ein Ereignis an `REQ` gestartet; nach Abschluss wird `CNF` ausgelöst.

Q_NumericValue

- **Typ**: `isobus::UT::Q::Q_NumericValue`
- **Parameter**:
  - `u16ObjId` = `OutputNumber_N3`
- **Ereigniseingänge**: `REQ` (Wert schreiben anfordern)
- **Ereignisausgänge**: (keine expliziten Ausgänge im Netzwerk)
- **Dateneingänge**: `u32NewValue` (UDINT-Wert, der geschrieben werden soll)
- **Datenausgänge**: keine
- **Funktionsweise**: Dieser FB schreibt einen neuen Wert (UDINT) auf das ISOBUS-Objekt mit der ID `OutputNumber_N3`. Bei einem Ereignis an `REQ` wird der anliegende Wert an den Bus übertragen.

Programmablauf und Verbindungen

Die Verschaltung erfolgt in einer einfachen Ereignis- und Datenkette:

1. **Ereignisverbindungen**:
   - `InputNumber_I3.IND` ⟶ `F_DWORD_TO_UDINT.REQ`
   - `F_DWORD_TO_UDINT.CNF` ⟶ `Q_NumericValue.REQ`
2. **Datenverbindungen**:
   - `InputNumber_I3.IN` ⟶ `F_DWORD_TO_UDINT.IN`
   - `F_DWORD_TO_UDINT.OUT` ⟶ `Q_NumericValue.u32NewValue`

Ablauf:

- Sobald das ISOBUS-Objekt `InputNumber_I3` einen neuen Wert vom Bus erhält (z. B. durch ein externes Steuergerät), wird das Ereignis `IND` am FB `InputNumber_I3` ausgelöst.
- Dieses Ereignis triggert die Konvertierung im FB `F_DWORD_TO_UDINT` (über dessen `REQ`-Eingang).
- Nach Abschluss der Konvertierung gibt `F_DWORD_TO_UDINT` das Ereignis `CNF` aus, welches den FB `Q_NumericValue` anstößt, den konvertierten Wert auf das Ausgangsobjekt `OutputNumber_N3` zu schreiben.

Somit wird der eingehende Wert nahezu verzögerungsfrei (nur durch die Konvertierung verzögert) an das Ausgangsobjekt weitergereicht.

Zusammenfassung

Die Übung **Uebung_011c** zeigt die grundlegende Datenflussverarbeitung mit 4diac unter Verwendung von ISOBUS-Funktionsbausteinen. Sie vermittelt:

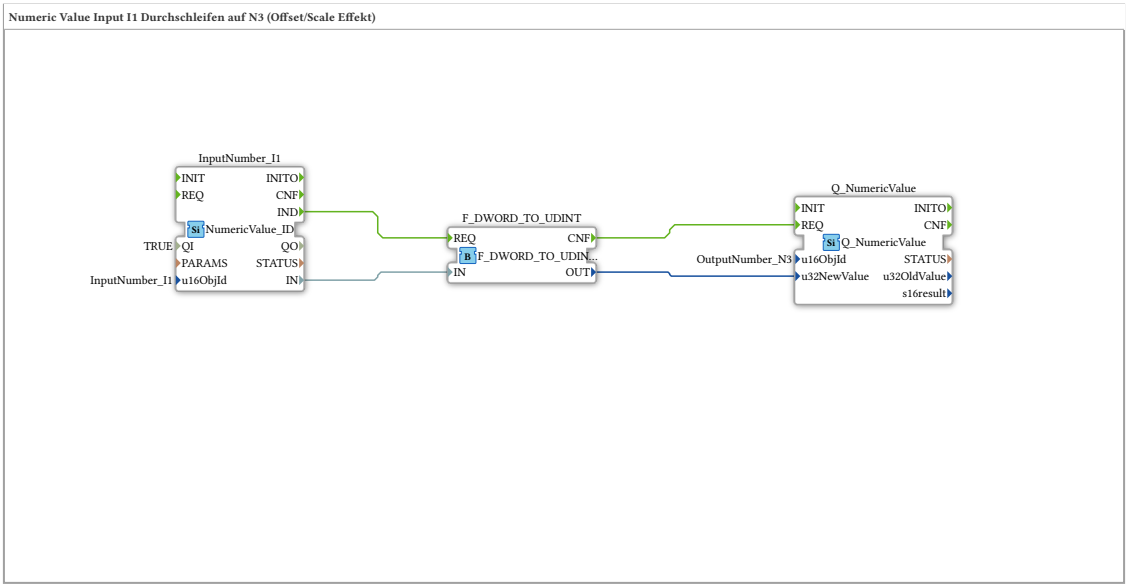
- Einlesen eines numerischen Werts von einem ISOBUS-Objekt,
- Datentypkonvertierung (DWORD ⟶ UDINT),
- Ausgabe des konvertierten Werts auf ein zweites Objekt,
- Ereignisgesteuerte Verkettung von FB-Instanzen.

Dieses einfache Durchschleifen kann als Basis für komplexere Signalverarbeitungsketten dienen, bei denen Werte zwischen verschiedenen Busteilnehmern ausgetauscht und gegebenenfalls skaliert oder umgewandelt werden müssen.

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

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

2.2.1.151 Uebung_011d: Numeric Value Input I1 Durchschleifen auf N3 (Offset/Scale Effekt)



• – ○ • – ○ • – ○ •

Einleitung

In dieser Übung wird ein numerischer Wert von einem Eingabegerät (I1) gelesen und unverändert an ein Ausgabegerät (N3) weitergeleitet. Durch die Verwendung eines Konvertierungsbausteins wird der eingehende 32-Bit-Wert von `DWORD` in `UDINT` umgewandelt. Diese Typumwandlung führt zu einem Offset-/Skalierungseffekt, der die Ausgabe gegenüber dem Rohwert verschiebt.

Ein Beispiel verdeutlicht den Effekt:

- Eingabe 100 000 ⟶ N3 zeigt 0,00
- Eingabe 50 000 ⟶ N3 zeigt −500,00

Die Übung zeigt die grundlegende Handhabung der `NumericValue`-Schnittstelle und die Auswirkungen von Datentypkonvertierungen.

Verwendete Funktionsbausteine (FBs)

Im Netzwerk der Übung werden drei Funktionsbausteine eingesetzt. Es sind keine Sub-Bausteine vorhanden.

Name	Typ	Parameter
`InputNumber_I1`	`isobus::UT::io::NumericValue::NumericValue`	`QIID = TRUE, u16ObjId = "InputNumber_I1"`
`F_DWORD_TO_UDINT`	`iec61131::conversion::F_DWORD_TO_UDINT`	(keine Parameter)
`Q_NumericValue`	`isobus::UT::Q::Q_NumericValue`	`u16ObjId = "OutputNumber_N3"`

- **`InputNumber_I1`** – Liest einen numerischen Rohwert vom Eingang I1 als `DWORD` (32-Bit) ein und stellt ihn am Datenausgang `IN` sowie ein Ereignis `IND` bei neuer Datenbereitstellung bereit.
- **`F_DWORD_TO_UDINT`** – Konvertiert den empfangenen `DWORD`-Wert in eine vorzeichenlose 32-Bit-Integer (`UDINT`). Die Umwandlung verändert die Interpretation der Bitfolge und erzeugt den beschriebenen Offset.
- **`Q_NumericValue`** – Nimmt den konvertierten `UDINT`-Wert über den Dateneingang `u32NewValue` entgegen und stellt ihn am Ausgang N3 dar. Der Baustein wird durch ein Ereignis an `REQ` getriggert.

Programmablauf und Verbindungen

Die Verarbeitung erfolgt ereignisgesteuert:

1. **Ereigniskette**
   - `InputNumber_I1` sendet bei einem neuen Eingabewert das Ereignis `IND`.
   - Dieses löst über eine Ereignisverbindung den `REQ`-Eingang von `F_DWORD_TO_UDINT` aus.
   - Nach der Konvertierung sendet `F_DWORD_TO_UDINT` das Ereignis `CNF`, das wiederum den `REQ`-Eingang von `Q_NumericValue` triggert.
2. **Datenverbindungen**
   - Der Ausgang `IN` von `InputNumber_I1` (Datentyp `DWORD`) ist mit dem Eingang `IN` von `F_DWORD_TO_UDINT` verbunden.
   - Der Ausgang `OUT` von `F_DWORD_TO_UDINT` (Datentyp `UDINT`) ist mit dem Dateneingang `u32NewValue` von `Q_NumericValue` verbunden.

Lernziele dieser Übung:

- Verständnis der Funktionsweise der `NumericValue`-Ein- und Ausgabebausteine.
- Erkennen des Einflusses von Datentypkonvertierungen (DWORD → UDINT) auf numerische Werte.
- Praktischer Umgang mit Ereignis- und Datenverbindungen in 4diac.
- Interpretation von Offset-/Skalierungseffekten durch Typumwandlung.

Die Übung erfordert Grundkenntnisse der 4diac-IDE und der isobus-Bibliothek. Sie kann direkt nach dem Import des Subapp-Typs im Netzwerkeditor gestartet werden – die Werte werden automatisch beim Verbinden mit einem entsprechenden Eingabegerät aktualisiert.

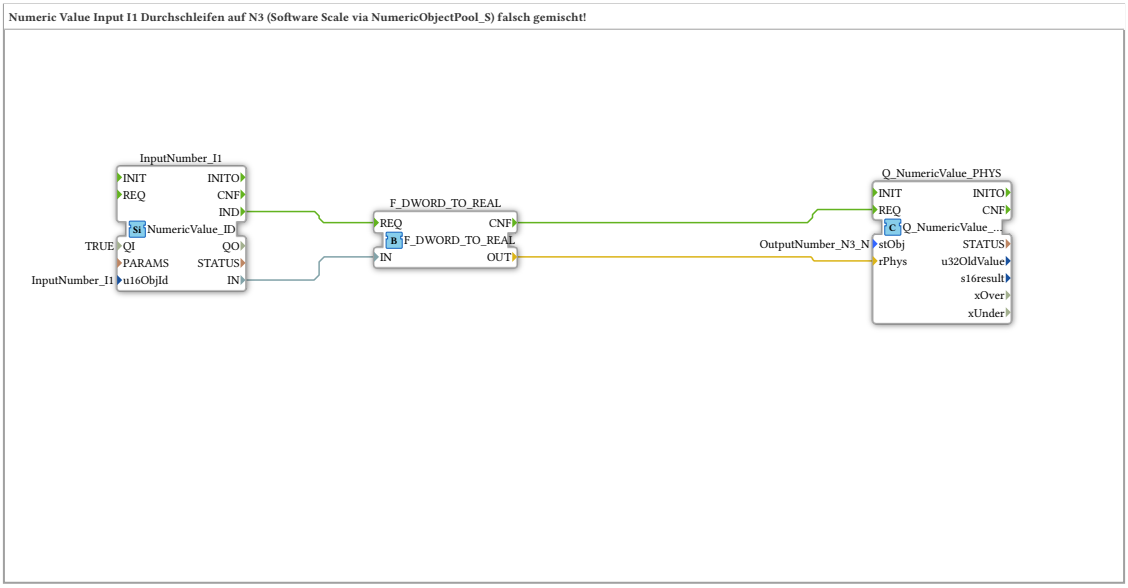
Zusammenfassung

Die Übung **Uebung_011d** demonstriert das Durchschleifen eines numerischen Werts von einem Eingang (I1) zu einem Ausgang (N3) unter Verwendung eines Konvertierungsbausteins. Die Umwandlung von `DWORD` in `UDINT` bewirkt einen Offset-/Skalierungseffekt, der die Ausgabe gegenüber dem Rohwert verschiebt. Durch die einfache Ereignis- und Datenverkettung wird das Grundprinzip der Datenverarbeitung mit den `NumericValue`-Bausteinen veranschaulicht und die Bedeutung der Datentypen hervorgehoben.

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

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

2.2.1.152 Uebung_011e_MIX: Numeric Value Input I1 Durchschleifen auf N3 (Software Scale via NumericObjectPool_S) falsch gemischt



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert ein **inkompatibles Zusammenspiel** zweier verschiedener Namespaces im isobus-Kontext. Das Ziel ist es, einen numerischen Wert von einem Eingang (I1) auf einen Ausgang (N3) durchzuschleifen, wobei bewusst eine Software-Skalierung über den `NumericObjectPool_S` verwendet wird – jedoch mit **falsch gemischten** Typen. Die Übung verdeutlicht die Probleme, die durch die Verwendung unterschiedlicher Datenrepräsentationen (Raw-Werte vs. physikalische Werte) entstehen können.

Im konkreten Beispiel: Eine Eingabe von `10` am Eingang I1 wird über den Baustein `F_RAW_TO_PHYS(I1)` (hier durch `F_DWORD_TO_REAL` ersetzt) zu `10.0` umgewandelt und dann an den Ausgang `N3` übergeben. Allerdings sind die Namespaces der verwendeten Pool-Objekte (`InputNumber_I1` und `OutputNumber_N3`) inkompatibel, was zu unerwartetem Verhalten führt – die Übung zeigt diese Fallstricke auf.

Verwendete Funktionsbausteine (FBs)

Die Übung besteht aus einer linearen Kette von drei Funktionsbausteinen (keine Sub-Bausteine):

Bausteinname	Typ	Beschreibung
`InputNumber_I1`	`isobus::UT::io::NumericValue::NumericValue_ID`	Liest einen numerischen Wert (DWORD) aus dem Pool `InputNumber_I1`. Der Parameter `u16ObjId` ist auf `"InputNumber_I1"` gesetzt, und der Qualifier `QI` ist `TRUE`. Der Ereignisausgang `IND` signalisiert einen neuen Wert am Eingang `IN`.
`F_DWORD_TO_REAL`	`iec61131::conversion::F_DWORD_TO_REAL`	Wandelt einen `DWORD`-Wert in einen `REAL`-Wert um (gemäß IEC 61131-3). Der Eingang `REQ` startet die Konvertierung, und nach Abschluss wird der Ausgang `CNF` getriggert.
`Q_NumericValue_PHYS`	`isobus::UT::Q::Q_NumericValue_PHYS`	Schreibt einen physikalischen `REAL`-Wert in das Ausgangsobjekt `OutputNumber_N3` (übernommen aus dem Pool `OutputNumber_N3`). Der Parameter `stObj` wird auf den entsprechenden String gesetzt. Der Baustein erwartet einen physikalischen Wert am Eingang `rPhys` und gibt diesen auf den Pool aus.

Programmablauf und Verbindungen

Der Ablauf ist ereignisgesteuert und erfolgt in drei Schritten:

1. **Eingabe lesen:** Wenn `InputNumber_I1` einen neuen Wert erhält (z. B. `10`), sendet es ein Ereignis über `IND` an den Konvertierungsbaustein `F_DWORD_TO_REAL.REQ`. Gleichzeitig wird der gelesene DWORD-Wert über die Datenverbindung `InputNumber_I1.IN` → `F_DWORD_TO_REAL.IN` übergeben.
2. **Konvertierung:** `F_DWORD_TO_REAL` wandelt den DWORD-Wert in einen REAL-Wert um (z. B. `10` → `10.0`). Nach Abschluss sendet es ein Ereignis über `CNF` an den Ausgangsbaustein `Q_NumericValue_PHYS.REQ`. Der konvertierte REAL-Wert wird über die Datenverbindung `F_DWORD_TO_REAL.OUT` → `Q_NumericValue_PHYS.rPhys` weitergegeben.
3. **Ausgabe schreiben:** `Q_NumericValue_PHYS` empfängt das Ereignis und schreibt den physikalischen REAL-Wert in das Pool-Objekt `OutputNumber_N3`. Am Bedienpanel erscheint dann beispielsweise `10.00`.

Besonderheit: In der Kommentarspalte wird darauf hingewiesen, dass die beiden Namespaces (`isobus::UT::io::NumericValue::NumericValue_ID` und `isobus::UT::Q::Q_NumericValue_PHYS`) **inkompatibel** sind. Der eingehende Wert wird zwar korrekt durchgeschleift, aber die semantische Zuordnung (Raw vs. Physical) wird verletzt, was zu Fehlinterpretationen in der Visualisierung führen kann. Die Übung soll daher für solche Inkompatibilitäten sensibilisieren.

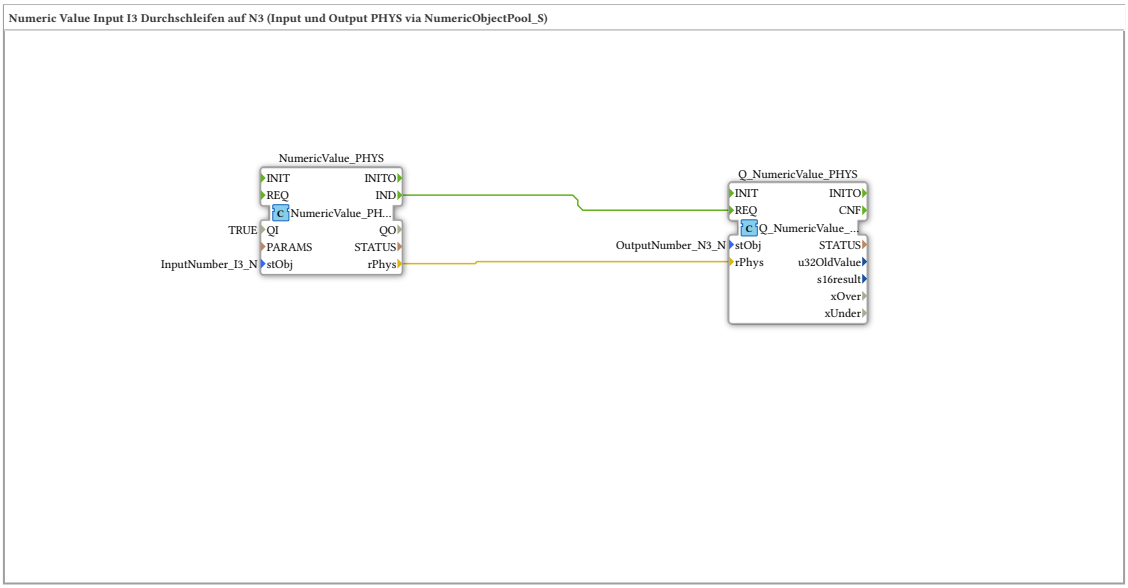
Zusammenfassung

Die Übung `Uebung_011e_MIX` zeigt, wie ein numerischer Wert vom Eingang `I1` über eine einfache Konvertierung (`DWORD → REAL`) zum Ausgang `N3` übertragen wird. Sie demonstriert jedoch bewusst eine **falsche Mischung** von Namespaces, die im produktiven Einsatz zu unerwarteten Ergebnissen führt. Ziel ist es, die Bedeutung der korrekten Auswahl von Pool-Objekten und den Umgang mit physikalischen vs. Rohdaten verständlich zu machen. Die Übung eignet sich für Einsteiger in die isobus-Konfiguration mit 4diac, die die Unterschiede zwischen verschiedenen Datentypen und Pools kennenlernen möchten.

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

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

2.2.1.153 Uebung_011f: Numeric Value Input I3 Durchschleifen auf N3 (Input und Output PHYS via NumericObjectPool_S)



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung demonstriert die direkte Durchschleifung eines numerischen Eingangswertes von der physikalischen Eingabe I3 zur physikalischen Ausgabe N3. Der Wert wird ohne Umrechnung oder Verarbeitung von einem `NumericValue_PHYS`-Baustein gelesen und über einen `Q_NumericValue_PHYS`-Baustein auf den Ausgang geschrieben. Die Verwendung eines `NumericObjectPool_S` (DefaultPool_Numeric) stellt die Verbindung zu den konkreten Ein-/Ausgangsobjekten her. Beispiel: **I3-Eingabe −500.00 → rPhys = −500.0 → Q_NumericValue_PHYS(N3) → N3 zeigt −500.00**

Verwendete Funktionsbausteine (FBs)

Sub-Baustein: Uebung_011f (SubAppType)

- **Typ**: SubAppType (eigene Subapplikation)
- **Verwendete interne FBs**:
  - **NumericValue_PHYS** *(Typ: `isobus::UT::io::NumericValue::NumericValue_PHYS`)*
    - Parameter:
      - `QI` = `TRUE`
      - `stObj` = `InputNumber_I3` (aus DefaultPool_Numeric)
    - Ereignisausgang: `IND`
    - Datenausgang: `rPhys` (Realwert)
  - **Q_NumericValue_PHYS** *(Typ: `isobus::UT::Q::Q_NumericValue_PHYS`)*
    - Parameter:
      - `stObj` = `OutputNumber_N3` (aus DefaultPool_Numeric)
    - Ereigniseingang: `REQ`

    - Dateneingang: `rPhys` (Realwert)
- **Funktionsweise**: Der Baustein `NumericValue_PHYS` liest den aktuellen physikalischen Wert des Eingangs I3. Sobald ein neuer Wert verfügbar ist, wird dies über das Ereignis `IND` signalisiert und der Wert am Ausgang `rPhys` bereitgestellt. Dieses Ereignis wird direkt mit dem Ereigniseingang `REQ` des Bausteins `Q_NumericValue_PHYS` verbunden. Gleichzeitig wird der Datenwert `rPhys` auf den entsprechenden Dateneingang des Ausgangsbausteins übertragen. Der Baustein `Q_NumericValue_PHYS` schreibt daraufhin den empfangenen Wert auf das physikalische Ausgangsobjekt N3. Die gesamte Subapplikation fungiert somit als eine transparente Durchschleifung von I3 nach N3.

Programmablauf und Verbindungen

- **Ereignisverbindung**: `NumericValue_PHYS.IND` ⟶ `Q_NumericValue_PHYS.REQ`
- **Datenverbindung**: `NumericValue_PHYS.rPhys` ⟶ `Q_NumericValue_PHYS.rPhys`

Der Ablauf ist rein ereignisgesteuert:

1. Der Eingangsbaustein erkennt eine Änderung an I3 und feuert `IND`.
2. Der Ausgangsbaustein wird über `REQ` getriggert, den aktuellen `rPhys`-Wert zu übernehmen und auf N3 auszugeben.
3. Da keine weiteren Verarbeitungsschritte erfolgen, wird der Wert eins zu eins übertragen.

Lernziele:

- Verständnis der direkten Verknüpfung von physikalischen Ein-/Ausgängen mittels `NumericValue_PHYS` und `Q_NumericValue_PHYS`.
- Kennenlernen der ereignisgesteuerten Datenweitergabe (ohne zusätzliche Logik).
- Anwendung des `NumericObjectPool_S` zur Konfiguration von I/O-Objekten.

Schwierigkeitsgrad: Einsteiger **Vorkenntnisse**: Grundlegende Kenntnisse der 4diac-IDE und des IEC 61499-Modells.

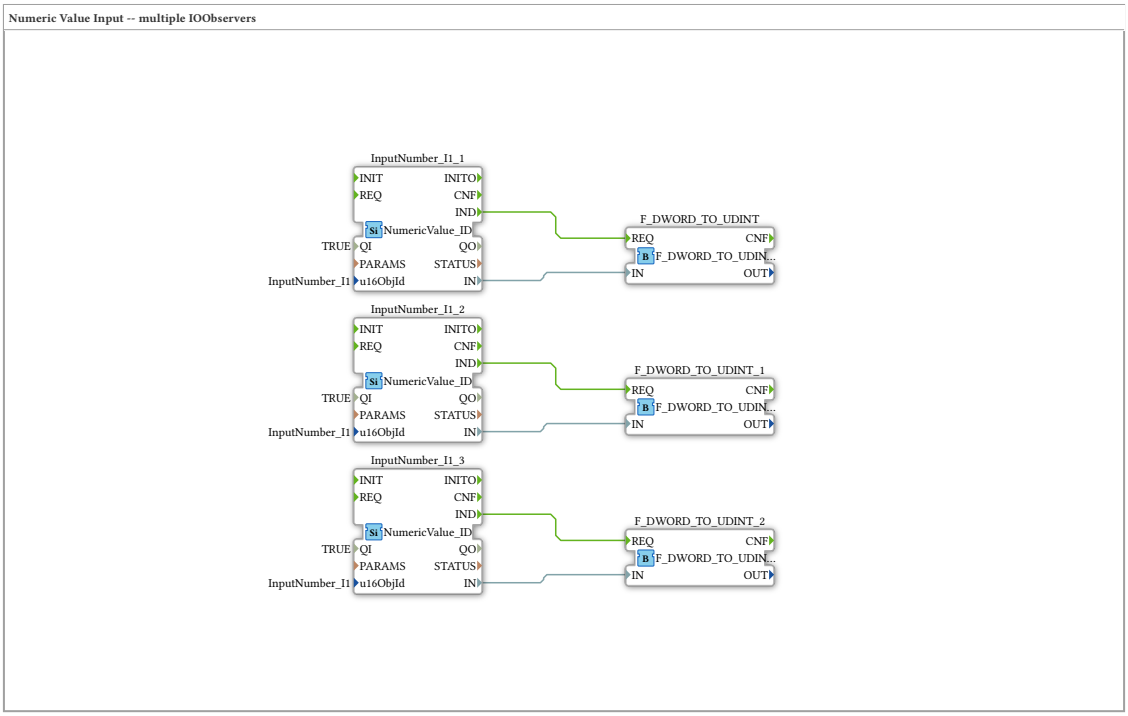
Zusammenfassung

Die Übung **Uebung_011f** realisiert eine einfache Durchschleifung eines numerischen Werts vom Eingang I3 zum Ausgang N3. Durch die Kombination von `NumericValue_PHYS` (lesend) und `Q_NumericValue_PHYS` (schreibend) wird eine klare Trennung zwischen I/O-Zugriff und Ereignissteuerung erreicht. Die Subapplikation ist als wiederverwendbarer Baustein konzipiert und kann direkt in größere Applikationen eingebunden werden.

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

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

2.2.1.154 Uebung_011g1: Numeric Value Input – multiple IOObservers



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert den parallelen Einsatz mehrerer `IOObserver` für einen gemeinsamen numerischen Eingangswert. Drei Instanzen des Funktionsbausteins `NumericValue_ID` überwachen denselben Objekt-Identifikator (`InputNumber_I1`). Die gelieferten `DWORD`-Werte werden jeweils mit einem Konvertierungsbaustein in den Typ `UDINT` umgewandelt. Die Übung vermittelt, wie mehrere Observer auf eine gemeinsame Datenquelle geschaltet werden können, ohne dass die Werte sich gegenseitig beeinflussen.

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: (keine – Hauptbaustein ist eine SubApp)

Die SubApp besteht direkt aus folgenden Funktionsbausteinen:

- **`NumericValue_ID`** (Typ: `isobus::UT::io::NumericValue::NumericValue_ID`)
  - **Parametrierung**:
    - `QI` = `TRUE` (Aktivierung)
    - `u16ObjId` = `InputNumber_I1` (Identifikator der numerischen Eingabe)
  - **Ereignisse**:
    - Ereignisausgang `IND` – signalisiert, dass ein neuer Wert anliegt
  - **Datenausgang**: `IN` (Typ `DWORD`) – aktueller Wert der beobachteten Objektinstanz
- **`F_DWORD_TO_UDINT`** (Typ: `iec61131::conversion::F_DWORD_TO_UDINT`)
  - **Keine weiteren Parameter**
  - **Ereigniseingang**: `REQ` – startet die Konvertierung
  - **Dateneingang**: `IN` vom Typ `DWORD`
  - **Datenausgang**: `OUT` vom Typ `UDINT` (unsigned double integer)
  - **Funktion**: Wandelt einen 32-Bit DWORD-Wert in einen vorzeichenlosen 32-Bit-Integer um.

Im Netzwerk sind drei identische Paare dieser Bausteine vorhanden:

Observer (NumericValue_ID)	Konverter (F_DWORD_TO_UDINT)
`InputNumber_I1_1`	`F_DWORD_TO_UDINT`

InputNumber_I1_2	F_DWORD_TO_UDINT_1
InputNumber_I1_3	F_DWORD_TO_UDINT_2

Programmablauf und Verbindungen

1. **Ereignisverbindungen** (EventConnections): Jeder Observer (IND) ist je mit dem Konverter (REQ) verbunden. Sobald ein neuer Wert vom ISOBUS-Gateway eintrifft, wird genau der zugehörige Konvertierungsvorgang ausgelöst.
2. **Datenverbindungen** (DataConnections): Der Datenausgang IN jedes Observers ist direkt mit dem Dateneingang IN des zugehörigen Konverters verbunden. Die drei Datenpfade sind vollständig voneinander isoliert; jeder Konverter arbeitet mit dem Wert des ihm zugeordneten Observers.
3. **Gemeinsame Quelle**: Alle drei NumericValue_ID-Bausteine beziehen ihre Daten vom selben ISOBUS-Objekt (InputNumber_I1). Die Observer können unabhängig voneinander arbeiten, da jeder eine eigene Kopie des aktuellen Werts erhält.
4. **Konvertierung**: Die Ausgänge OUT der drei F_DWORD_TO_UDINT-Bausteine stellen das gleiche numerische Signal als UDINT dar – für unterschiedliche Verbraucher innerhalb der Anwendung.

Lernziele

- Verständnis für die parallele Überwachung einer einzigen ISOBUS-Variable mit mehreren IOObserver-Instanzen.
- Anwendung von Typkonvertierungen (DWORD ⟶ UDINT) in 4diac.
- Fehlervermeidung durch getrennte Signalwege (kein Daten-Overlap).

Schwierigkeitsgrad

Einfach – grundlegende Funktionsbausteine und einfache Verdrahtung. Vorausgesetzt werden Basiskenntnisse in 4diac und im Umgang mit ISOBUS-Datenobjekten.

Zusammenfassung

Die Übung Uebung_011g1 zeigt ein Muster, bei dem ein einzelner numerischer Eingang (InputNumber_I1) von drei unabhängigen Observer-Bausteinen gelesen wird. Jeder Observer löst einen eigenen Konvertierungs-FB (DWORD ⟶ UDINT) aus, sodass drei identische, aber isolierte Signalpfade entstehen. Dies ist nützlich, wenn der Wert an mehreren Stellen in der Steuerung benötigt wird und jede Stelle eine eigene, störungsfreie Datenkopie erfordert. Die einfache Struktur eignet sich hervorragend für Einstiegsübungen zur parallelen Datenverarbeitung mit 4diac.

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

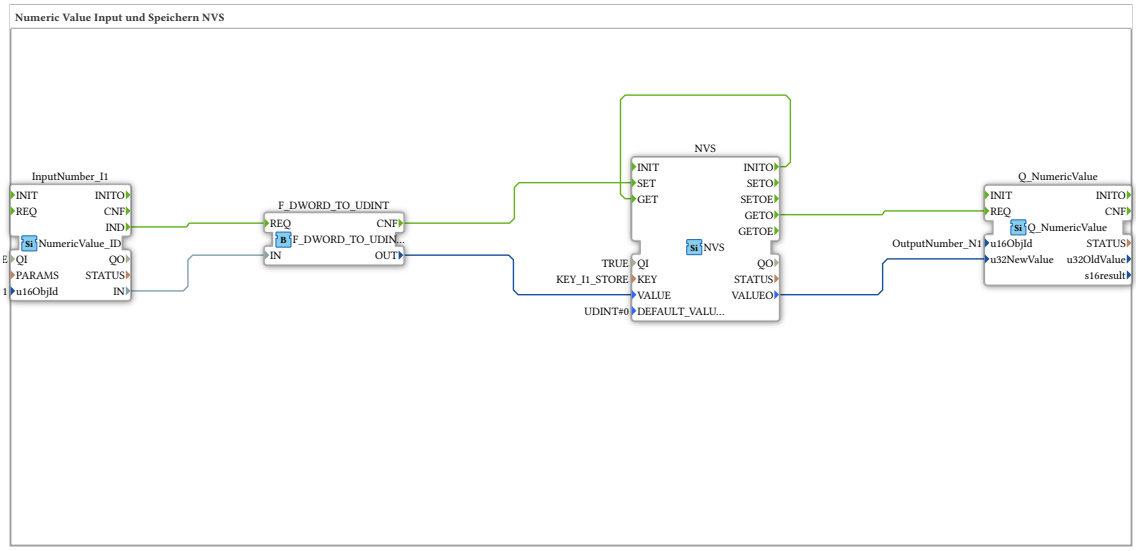
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.2.1.155 Uebung_012: Numeric Value Input und Speichern NVS

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung Uebung_012. Hier wird gezeigt, wie numerische Werte nicht nur angezeigt, sondern auch stromausfallsicher in der Steuerung (NVS - Non Volatile Storage) gespeichert werden.

🎧 Podcast



- Amazon Pizza-Regel bis IKEA-Effekt: 12 verblüffend einfache Ideen hinter riesigem Geschäftserfolg

---

Ziel der Übung

Erlernen der persistenten Datenspeicherung. Es wird demonstriert, wie ein am Terminal eingegebener Wert im internen Flash-Speicher der Steuerung abgelegt und beim Neustart automatisch wieder geladen und angezeigt wird.

Beschreibung und Komponenten

Die Subapplikation `Uebung_012.SUB` verbindet Eingabe, Speicherung und Anzeige zu einem geschlossenen Kreislauf.

Funktionsbausteine (FBs)

- **`InputNumber_I1`**: Numerisches Eingabefeld am Terminal.
- **`NVS`**: Typ `logiBUS::storage::esp32_nvs::NVS`. Dieser Baustein verwaltet den Zugriff auf den nichtflüchtigen Speicher. Er speichert Werte unter einem eindeutigen `KEY` ab.
- **`CbVtStatus`**: Ein Statusbaustein des Terminals. Er feuert ein Ereignis (`IND`), wenn das Terminal neu startet oder die Verbindung wiederhergestellt wird.
- **`Q_NumericValue`**: Die numerische Anzeige am Terminal.

Funktionsweise

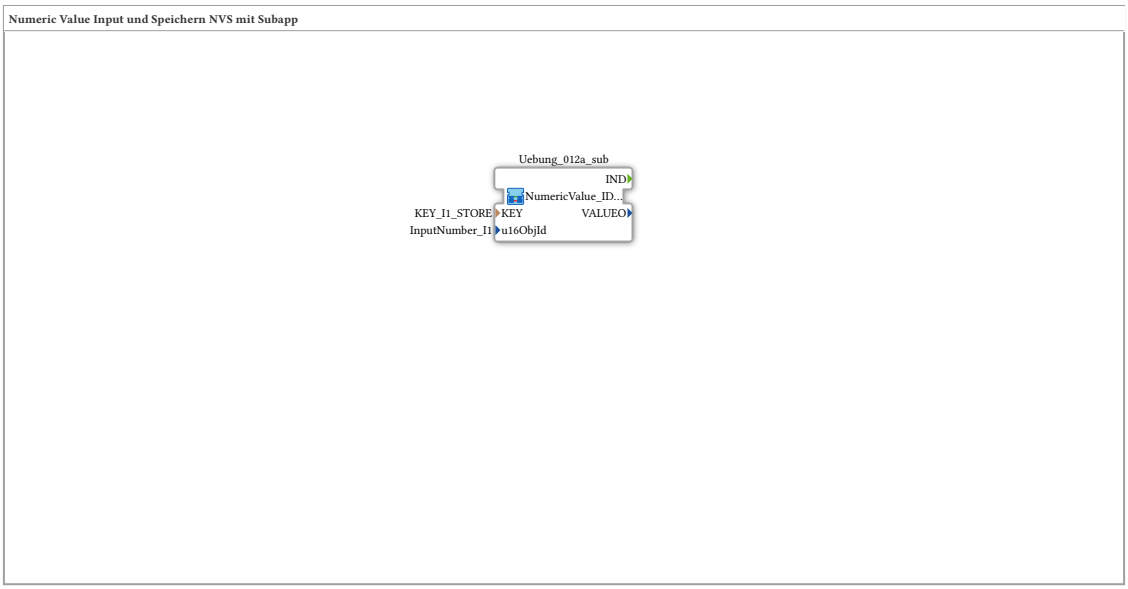
Der Prozess deckt drei Szenarien ab:

1. **Speichern**: Gibt der Nutzer einen Wert ein (`IND`), wird dieser konvertiert und per `NVS.SET` dauerhaft gespeichert.
2. **Laden beim Start**: Nach dem Booten sendet der Speicherbaustein ein `INITO`-Event, welches sofort einen Lese-Vorgang (`GET`) auslöst. Der gespeicherte Wert wird geladen und an die Anzeige gesendet.
3. **Refresh bei Verbindung**: Falls das Terminal während des Betriebs kurzzeitig getrennt wird, sorgt `CbVtStatus.IND` dafür, dass der aktuelle Wert erneut an das Terminal gesendet wird, sobald es wieder online ist.

Anwendungsbeispiel

Konfigurations-Parameter: Ein Landwirt stellt die Arbeitsbreite seines Gerätes einmalig am Terminal ein. Dank NVS-Speicherung muss er diesen Wert nicht bei jedem morgendlichen Start der Maschine neu eingeben; die Steuerung „erinnert“ sich an die letzte Einstellung.

2.2.1.156 Uebung_012a: Numeric Value Input und Speichern NVS mit Subapp



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_012a`. Hier wird die persistente Speicherung aus Übung 012 in eine wiederverwendbare Sub-Applikation gekapselt.

Übersicht

Die Subapplikation `Uebung_012a.SUB` nutzt den Sub-App-Typ `Uebung_012a_sub`, um die Speicher-Logik modular bereitzustellen. Der Baustein `CbVtStatus` bleibt auf der obersten Ebene, um die gesamte Seite bei Bedarf zu aktualisieren.

Typisierte Sub-Applikation: `Uebung_012a_sub`

Dieser Baustein bündelt die Eingabe via `NumericValue_ID`, die Konvertierung, den NVS-Zugriff und die Anzeige-Rückführung. Er stellt Schnittstellen für den Speicher-Schlüssel (`KEY`) und die Objekt-ID (`u16ObjId`) zur Verfügung.

Dies ermöglicht es, viele verschiedene Einstellwerte (z.B. Druck, Durchfluss, Zeit) sehr schnell und übersichtlich in das Programm zu integrieren, ohne jedes Mal das komplexe Netzwerk aus Übung 012 neu zeichnen zu müssen.

2.2.1.157 Uebung_012a_sub

Hinweis

Diese Übung existiert nicht mehr als eigene Übung. Der Baustein wurde in die Bibliothek verschoben und heißt jetzt NumericValue_ID_TO_NVS. Beschreibung, Schnittstelle und Netzwerk stehen dort.

2.2.1.158 Uebung_012b: Numeric Value Input und Speichern INI

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_012b`. Hier wird eine alternative Methode zur Speicherung von Daten vorgestellt: Die Verwendung von INI-Dateien.

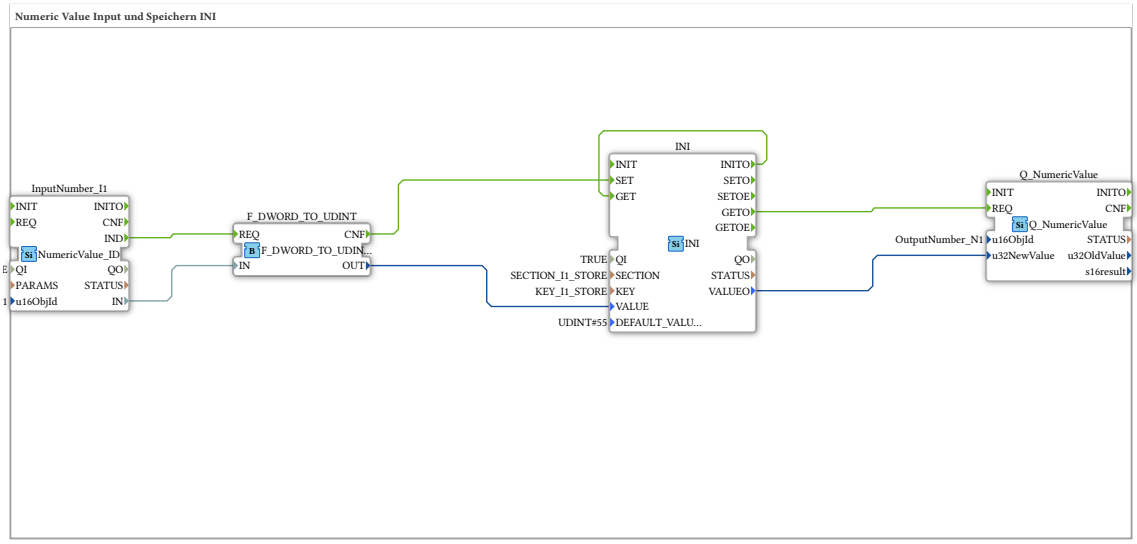
Ziel der Übung

Verwendung des `INI` Bausteins zur strukturierten Datenspeicherung. Im Gegensatz zum einfachen NVS-Key-Value-Speicher erlaubt das INI-Format eine Gliederung in Sektionen und Schlüssel, was bei großen Datenmengen übersichtlicher ist.

Beschreibung und Komponenten

Die Subapplikation `Uebung_012b.SUB` nutzt einen INI-Speicher-Baustein.

Funktionsbausteine (FBs)



- **`INI`**: Typ `eclipse4diac::storage::INI`. Dieser Baustein speichert Werte in einer dateibasierten Struktur ab. Er benötigt zusätzlich zum `KEY` eine `SECTION`.
- **Parameter**:
  - `SECTION`: „SECTION_I1_STORE“
  - `KEY`: „KEY_I1_STORE“
  - `DEFAULT_VALUE`: 55 (wird geladen, falls noch keine Datei existiert).

Funktionsweise

Die Logik entspricht ansonsten der Übung 012:

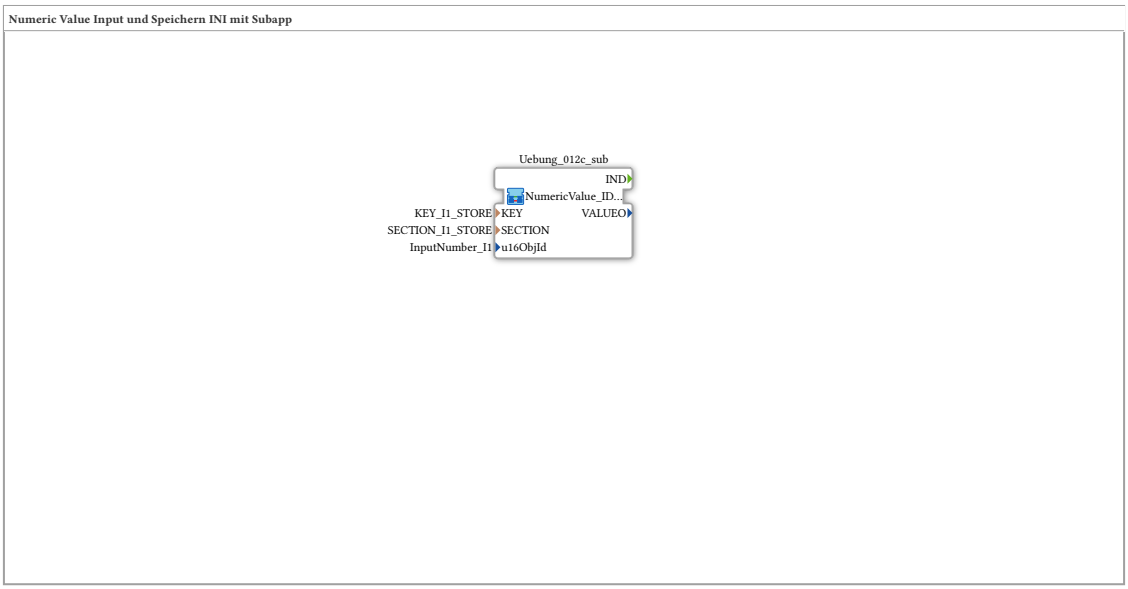
1. **Schreiben**: `InputNumber -> REQ -> INI.SET`.
2. **Lesen**: `INITO -> INI.GET -> Q_NumericValue`.
3. **Refresh**: `CbVtStatus -> Q_NumericValue`.

INI-Dateien sind besonders nützlich, wenn Parameter extern (z.B. über einen PC oder Web-Interface) ausgelesen oder editiert werden sollen, da sie in einem für Menschen lesbaren Textformat vorliegen.

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

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

2.2.1.159 Uebung_012c: Numeric Value Input und Speichern INI mit Subapp





Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_012c`. Hier wird demonstriert, dass auch Text-Informationen persistent gespeichert werden können.

Ziel der Übung

Umgang mit dem Datentyp `STRING` bei der Speicherung.

Beschreibung und Komponenten

In `Uebung_012c.SUB` wird ein String-Wert über einen INI-Baustein verwaltet.

Funktionsbausteine (FBs)

- **`INI`**: Hier konfiguriert für den Datentyp `STRING`.
- **`Q_StringValue`**: Ein ISOBUS-Ausgangsbaustein, der speziell für die Anzeige von Texten (anstelle von Zahlen) gedacht ist.

Funktionsweise

Der Ablauf ist identisch zur numerischen Speicherung, jedoch fließen über die Datenverbindungen nun Zeichenketten. Beim Systemstart wird der gespeicherte Text (z.B. ein Gerätename oder ein Benutzerprofil) geladen und an das Terminal gesendet.

Anwendungsbeispiel

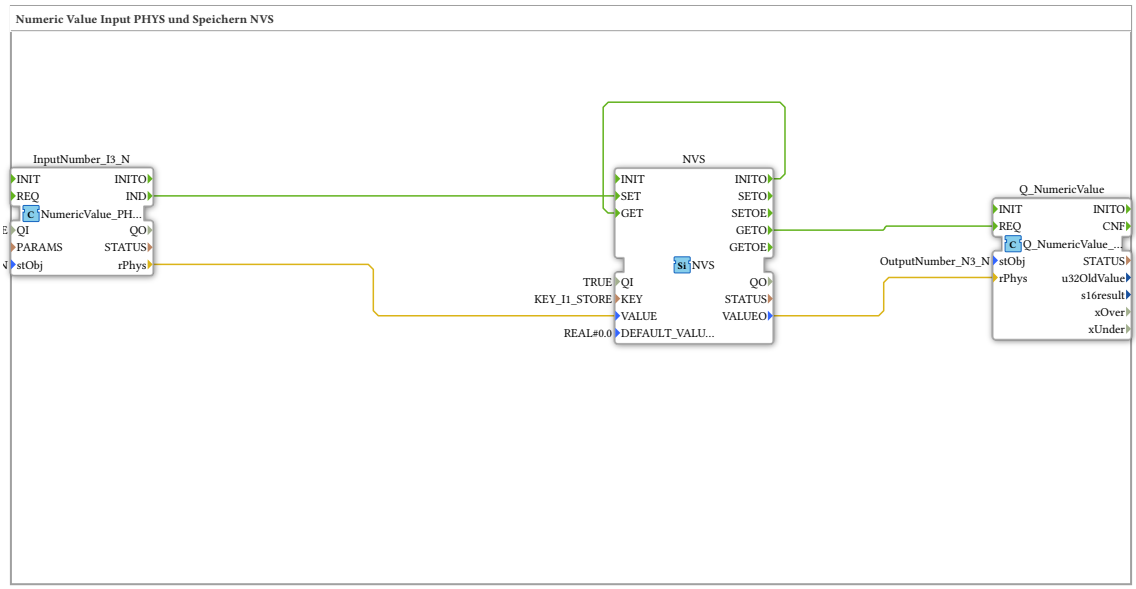
Personalisierung: Der Nutzer kann am Terminal seinen Namen oder den Namen des Feldes eingeben, auf dem er gerade arbeitet. Diese Information wird dauerhaft gespeichert und kann später für Protokollierungszwecke (z.B. Task Controller) genutzt werden.

2.2.1.160 Uebung_012c_sub

Hinweis

Diese Übung existiert nicht mehr als eigene Übung. Der Baustein wurde in die Bibliothek verschoben und heißt jetzt NumericValue_ID_TO_INI. Beschreibung, Schnittstelle und Netzwerk stehen dort.

2.2.1.161 Uebung_012d: Numeric Value Input PHYS und Speichern NVS



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert die Erfassung eines numerischen Werts (Physical Value) über einen Eingabebaustein, das Speichern des Werts im nichtflüchtigen Speicher (NVS) sowie das Auslesen und Bereitstellen des gespeicherten Werts über einen Ausgabebaustein. Ziel ist es, den Ablauf von Datenerfassung, persistenter Speicherung und Rückgabe zu verstehen.

Verwendete Funktionsbausteine (FBs)

Die Übung verwendet drei Funktionsbausteine, die innerhalb der Subapplikation vernetzt sind:

1. **InputNumber_I3** – Typ: `isobus::UT::io::NumericValue::NumericValue_PHYS`
   - Parameter:
     - `QI` = `TRUE`
     - `stObj` = `InputNumber_I3`
   - Funktion: Stellt die physikalische Eingabe eines numerischen Werts bereit. Der Wert wird über den Datenausgang `rPhys` ausgegeben, sobald ein Ereignis am Eingang `IND` anliegt.
2. **NVS** – Typ: `logiBUS::storage::esp32_nvs::NVS`
   - Parameter:
     - `QI` = `TRUE`
     - `KEY` = `KEY_I1_STORE` (importiert aus `Uebungen::const::NVS::NVS_Keys`)
     - `DEFAULT_VALUE` = `REAL#0.0`
   - Funktion: Nichtflüchtiger Speicherbaustein. Er speichert einen Wert unter einem vorgegebenen Schlüssel (`KEY`) und kann diesen bei Bedarf wieder auslesen. Die Ereignisse `SET` und `GET` steuern das Speichern bzw. Lesen. Die Ausgänge `VALUEO` liefern den gespeicherten Wert.
3. **Q_NumericValue** – Typ: `isobus::UT::Q::Q_NumericValue_PHYS`
   - Parameter:
     - `stObj` = `OutputNumber_N3`
   - Funktion: Stellt einen numerischen Wert als physikalischen Ausgang bereit. Der Wert wird über den Dateneingang `rPhys` entgegengenommen und bei einem Ereignis am Eingang `REQ` ausgegeben.

Sub-Bausteine

Die Übung selbst ist als SubAppType definiert und enthält keine weiteren Sub-Bausteine. Die drei oben genannten FBs sind direkt im Netzwerk verbunden.

Programmablauf und Verbindungen

Der Ablauf ist wie folgt:

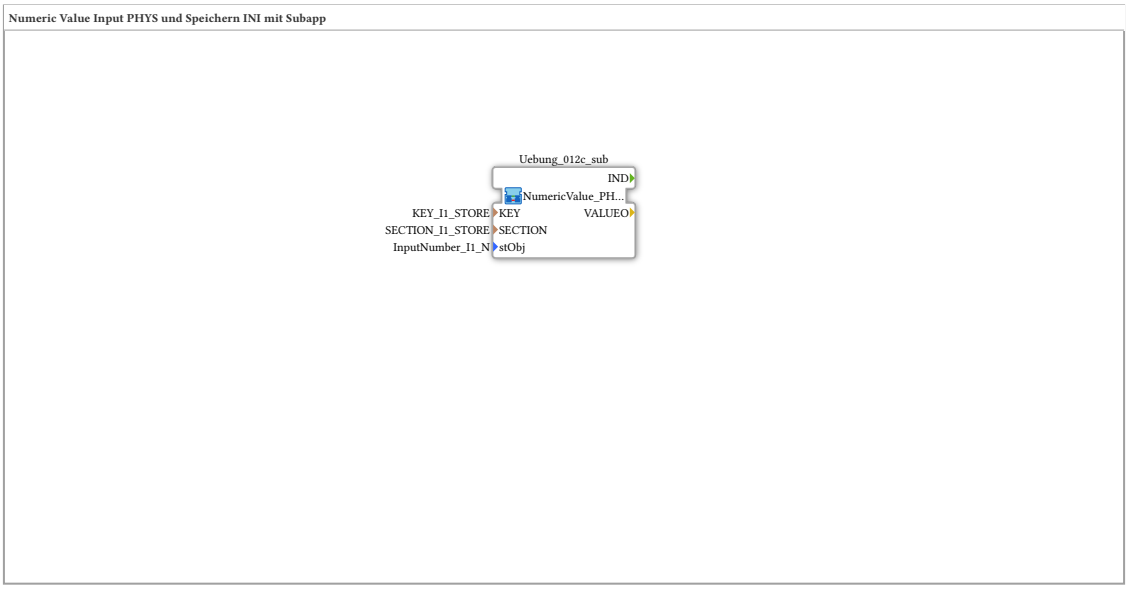
1. **Eingabe und Speichern**
   - Wenn ein neuer physikalischer Wert an `InputNumber_I3` anliegt, erzeugt dieser ein Ereignis am Ausgang `IND`.
   - Dieses Ereignis wird über eine Eventverbindung an den Eingang `SET` des NVS-Bausteins weitergeleitet.
   - Gleichzeitig wird der Datenwert von `InputNumber_I3.rPhys` über eine Datenverbindung an den NVS-Dateneingang `VALUE` übergeben.
   - Der NVS speichert den Wert unter dem Schlüssel `KEY_I1_STORE` im nichtflüchtigen Speicher.
2. **Initialisieren und Auslesen**
   - Nach dem Start (oder nach einer Initialisierung) erzeugt der NVS-Baustein ein Ereignis am Ausgang `INITO`.
   - Dieses Ereignis wird über eine Eventverbindung an den Eingang `GET` des NVS zurückgeführt (Selbsttriggerung).
   - Der NVS liest daraufhin den gespeicherten Wert aus und legt ihn am Datenausgang `VALUEO` an.
   - Gleichzeitig wird ein Ereignis am Ausgang `GETO` erzeugt.
3. **Ausgabe**
   - Das Ereignis `NVS.GETO` wird an den Eingang `REQ` des Ausgabebausteins `Q_NumericValue` weitergeleitet.
   - Der Datenwert `NVS.VALUEO` wird über eine Datenverbindung an den Eingang `Q_NumericValue.rPhys` übergeben.
   - `Q_NumericValue` stellt den Wert daraufhin am physikalischen Ausgang `OutputNumber_N3` bereit.

Zusammengefasst: Jeder eingehende Wert wird sofort gespeichert, und beim Systemstart wird der zuletzt gespeicherte Wert automatisch ausgegeben. Die Übung eignet sich für Einsteiger, die das Zusammenspiel von Ein-/Ausgabebausteinen mit einem nichtflüchtigen Speicher kennenlernen möchten.

Zusammenfassung

Die Übung `Uebung_012d` implementiert eine robuste Speicher- und Wiederherstellungsfunktion für einen physikalischen Zahlenwert. Durch die Kopplung eines Eingabebausteins (`NumericValue_PHYS`) mit einem NVS-Baustein und einem Ausgabebaustein (`Q_NumericValue_PHYS`) wird sichergestellt, dass der letzte Wert auch nach einem Neustart verfügbar bleibt. Der Ablauf ist einfach und nachvollziehbar: Wert erfassen → speichern → initial auslesen → ausgeben.

2.2.1.162 Uebung_012e: Numeric Value Input PHYS und Speichern INI mit Subapp



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung demonstriert das Einlesen eines numerischen Werts über einen physikalischen Eingangsbaustein (`NumericValue_PHYS`) und das dauerhafte Speichern des Werts mittels eines INI-Datei-Bausteins. Die gesamte Logik ist in einer Subapp gekapselt, die als wiederverwendbare Komponente dient. Ziel ist es, einen Wert über eine objektbasierte Konfiguration zu erfassen, zwischenzuspeichern und bei Bedarf wieder auszulesen.

Verwendete Funktionsbausteine (FBs)

Haupt-Subapp: `Uebung_012e`

- Instanziiert die Subapp `Uebung_012e_sub` mit vorgegebenen Parametern für Schlüssel, Abschnitt und Objekt.
- Die Haupt-Subapp hat keine eigenen Ein-/Ausgänge; das Verhalten wird vollständig durch den Sub-Baustein definiert.

Sub-Bausteine: `Uebung_012e_sub`

- **Typ**: `SubAppType`
- **Verwendete interne FBs**:
  - **`NumericValue_PHYS`**: `isobus::UT::io::NumericValue::NumericValue_PHYS`
    - Parameter: `QI` = `TRUE`
    - Ereignisausgang: `IND` (wird nach erfolgreicher Werteerfassung ausgelöst)
    - Dateneingang: `stObj` (Objektkonfiguration vom Typ `NumericObjectPool_S`)
    - Datenausgang: `rPhys` (physikalischer Wert als `REAL`)
  - **`INI`**: `eclipse4diac::storage::INI`
    - Parameter:
      - `QI` = `TRUE`
      - `DEFAULT_VALUE` = `REAL#0.0`
    - Ereigniseingänge: `SET`, `GET`, `INIT`
    - Ereignisausgänge: `SETO`, `GETO`, `INITO`
    - Dateneingänge: `KEY` (`STRING`), `SECTION` (`STRING`), `VALUE` (`REAL`)
    - Datenausgang: `VALUEO` (`REAL`)
  - **`Q_NumericValue_PHYS`**: `isobus::UT::Q::Q_NumericValue_PHYS`
    - Ereigniseingang: `REQ`
    - Dateneingänge: `stObj` (Objektkonfiguration), `rPhys` (Wert)
    - Funktion: Qualifiziert den physikalischen Wert (z. B. zur Skalierung oder Offsetsetzung)
- **Funktionsweise**:
  1. Der Baustein `NumericValue_PHYS` liest gemäß der Objektkonfiguration (`stObj`) einen Wert und gibt diesen als `rPhys` aus. Nach erfolgreicher Erfassung wird das Ereignis `IND` gesendet.
  2. Das Ereignis `IND` löst den `SET`-Eingang des `INI`-Bausteins aus. Dieser speichert den Wert unter dem angegebenen `KEY` in der `SECTION` der INI-Datei.
  3. Nach dem Speichern wird `SETO` ausgelöst, welches über den Ausgang `IND` der Subapp weitergereicht wird.
  4. Gleichzeitig startet die Initialisierungskette: `INITO` von `INI` wird mit `GET` verbunden, sodass nach dem Start der Subapp der gespeicherte Wert ausgelesen wird.
  5. Der ausgelesene Wert (`VALUEO`) wird an den Baustein `Q_NumericValue_PHYS` übergeben, der ihn mit der Objektkonfiguration qualifiziert und an den Ausgang `VALUEO` der Subapp weiterleitet.

Programmablauf und Verbindungen

Die Subapp arbeitet in zwei Hauptphasen:

1. **Initialisierung (Start)**:
   - Das Ereignis `INITO` des `INI`-Bausteins wird intern mit dem `GET`-Eingang verbunden.
   - Dadurch wird sofort der gespeicherte Wert aus der INI-Datei gelesen.
   - Der gelesene Wert wird über `VALUEO` an `Q_NumericValue_PHYS` und an den Ausgang der Subapp weitergegeben.
   - Das Ereignis `GETO` wird ebenfalls an den Ausgang `IND` der Subapp gesendet.
2. **Zyklischer Betrieb (bei Wertänderung)**:
   - `NumericValue_PHYS` erfasst bei Aktivierung einen neuen physikalischen Wert.

- Dieser Wert wird über `rPhys` an den `VALUE`-Eingang des `INI`-Bausteins gelegt.
- Das Ereignis `IND` von `NumericValue_PHYS` triggert den `SET`-Eingang von `INI`, wodurch der Wert gespeichert wird.
- Nach erfolgreichem Speichern wird `SETO` ausgelöst, das als `IND` der Subapp nach außen gegeben wird.

Datenverbindungen:

- Die Objektkonfiguration (`stObj`) wird von außen an `NumericValue_PHYS` und an `Q_NumericValue_PHYS` übergeben.
- Die Schlüssel- und Abschnittsnamen (`KEY`, `SECTION`) werden von außen an den `INI`-Baustein übergeben.
- Der gespeicherte Wert (`VALUEO`) wird sowohl an `Q_NumericValue_PHYS` als auch direkt an den Ausgang `VALUEO` der Subapp geführt.

Zusammenfassung

Die Übung vermittelt den Umgang mit physikalischen Werteingaben und persistenter Datenspeicherung in 4diac. Sie zeigt, wie man mit der Subapp-Technik eine wiederverwendbare Komponente erstellt, die:

- einen numerischen Wert über ein Objekt einliest,
- diesen Wert in einer INI-Datei speichert,
- den gespeicherten Wert bei Systemstart wiederherstellt und
- den Wert über einen Qualifier-Baustein prozessiert.

Durch die Kombination der Bausteine `NumericValue_PHYS`, `INI` und `Q_NumericValue_PHYS` wird ein typisches Szenario aus der Automatisierungstechnik (Erfassen, Speichern und Wiederherstellen von Parametern) abgebildet.

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

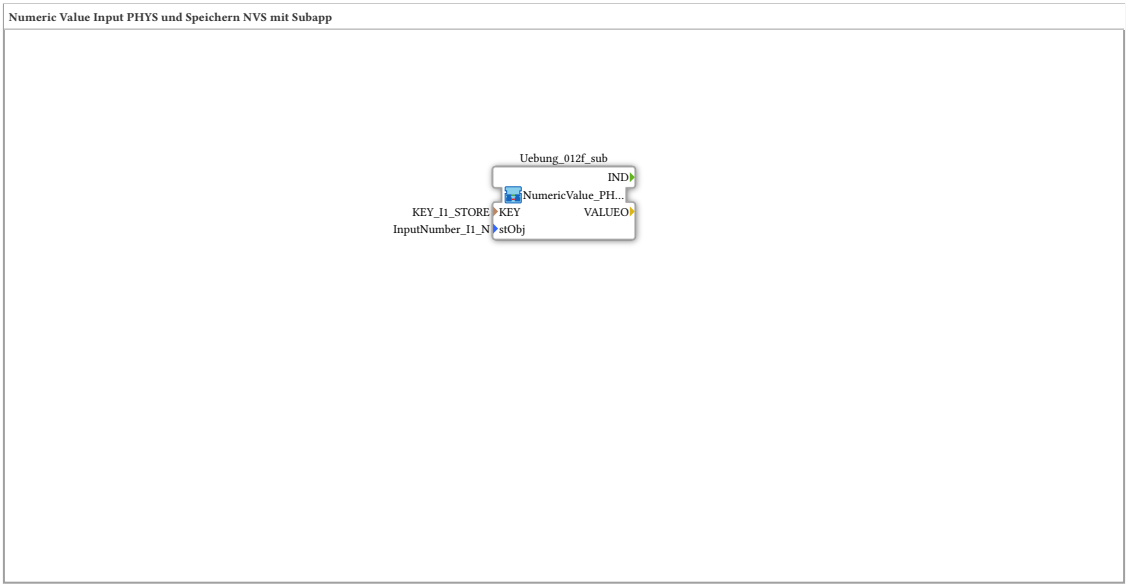
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.2.1.163 Uebung_012e_sub

Hinweis

Diese Übung existiert nicht mehr als eigene Übung. Der Baustein wurde in die Bibliothek verschoben und heißt jetzt NumericValue_PHYS_TO_INI. Beschreibung, Schnittstelle und Netzwerk stehen dort.

2.2.1.164 Uebung_012f: Numeric Value Input PHYS und Speichern NVS mit Subapp



• – ○ • – ○ • – ○ •

Einleitung

Die Übung demonstriert die Verarbeitung eines physikalischen numerischen Werts (PHYS) und dessen dauerhafte Speicherung in einem nichtflüchtigen Speicher (NVS). Der Wert wird über einen Eingangsobjekt-Pool konfiguriert, in den NVS geschrieben und kann anschließend über einen Qualifier-Baustein ausgegeben werden. Die Übung ist als geschachtelte SubApplikation realisiert, wobei der innere Sub-Baustein die gesamte Logik enthält.

Verwendete Funktionsbausteine (FBs)

Die Übung besteht aus einem Haupt-Sub-Baustein, der einen weiteren Sub-Baustein enthält. Dieser innere Sub-Baustein kapselt die eigentliche Verarbeitungslogik mit drei internen FBs.

Sub-Baustein: Uebung_012f_sub

- **Typ**: `Uebungen::Uebung_012f_sub`
- **Schnittstelle**:
  - Ereignisausgang: `IND` (Event)
  - Dateneingänge: `KEY` (STRING), `stObj` (NumericObjectPool_S)
  - Datenausgang: `VALUEO` (REAL)
- **Verwendete interne FBs**:
  - **NumericValue_PHYS** (`isobus::UT::io::NumericValue::NumericValue_PHYS`)
    - Parameter: `QI` = `TRUE`
    - Ereignisausgang: `IND` ⟶ wird bei gültigem Wert ausgelöst
    - Dateneingang: `stObj` (NumericObjectPool_S – Konfiguration des physikalischen Objekts)
    - Datenausgang: `rPhys` (REAL) – der gelesene physikalische Wert
  - **NVS** (`logiBUS::storage::esp32_nvs::NVS`)
    - Parameter: `QI` = `TRUE`, `DEFAULT_VALUE` = `REAL#0.0`
    - Ereigniseingänge: `INIT`, `SET`, `GET`
    - Ereignisausgänge: `INITO`, `SETO`, `GETO`
    - Dateneingänge: `KEY` (STRING), `VALUE` (REAL)
    - Datenausgang: `VALUEO` (REAL) – der gelesene oder geschriebene Wert
  - **Q_NumericValue_PHYS** (`isobus::UT::Q::Q_NumericValue_PHYS`)
    - Ereigniseingang: `REQ`
    - Dateneingänge: `stObj` (NumericObjectPool_S), `rPhys` (REAL)
    - Datenausgang: (keine im XML sichtbar, dient der Qualifikation/Weitergabe)
- **Funktionsweise**: Der Sub-Baustein `Uebung_012f_sub` empfängt die Konfigurationsdaten `KEY` und `stObj` von außen. Nach der Initialisierung des NVS-Bausteins (Ereignis `INITO`) wird sofort ein `GET` ausgelöst, um den zuletzt gespeicherten Wert zu laden. Bei einem gültigen physikalischen Wert (Ereignis `IND` vom `NumericValue_PHYS`) wird dieser mit dem aktuellen `rPhys` über `SET` im NVS gespeichert. Der geladene oder gespeicherte Wert wird über den Qualifier-Baustein `Q_NumericValue_PHYS` aufbereitet und am Ausgang `VALUEO` bereitgestellt. Die Ereignisse `SETO` und `GETO` werden zum äußeren `IND`-Ausgang weitergeleitet.

Programmablauf und Verbindungen

Der Ablauf ist ereignisgesteuert:

1. **Initialisierung**: Beim Start erhält der NVS-Baustein ein `INIT`-Ereignis (nicht im Netzwerk sichtbar, aber implizit durch die Umgebung). Der Baustein quittiert mit `INITO` und löst sofort ein `GET` aus (Verbindung `NVS.INITO -> NVS.GET`). Dadurch wird der unter dem übergebenen `KEY` gespeicherte Wert aus dem NVS gelesen.
2. **Wertaktualisierung**: Sobald der `NumericValue_PHYS`-Baustein einen gültigen physikalischen Wert erfasst hat, sendet er ein `IND`-Ereignis. Dieses triggert den `SET`-Eingang des NVS, wodurch der aktuelle `rPhys`-Wert unter dem angegebenen `KEY` gespeichert wird.
3. **Ausgabe**: Sowohl nach einem `GET` (bei Initialisierung) als auch nach einem `SET` (nach Speicherung) wird das Ergebnis über `NVS.VALUEO` an den `Q_NumericValue_PHYS`-Baustein und direkt an den Ausgang `VALUEO` weitergeleitet. Der Qualifier-Baustein bereitet den Wert entsprechend der Konfiguration (`stObj`) auf.
4. **Signalweitergabe**: Die Ereignisausgänge `SETO` und `GETO` des NVS werden zum äußeren Ausgang `IND` durchgereicht, sodass die übergeordnete Ebene über Änderungen informiert wird.

Datenverbindungen:

- `stObj` (extern) ⟶ `NumericValue_PHYS.stObj` (Konfiguration des physikalischen Objekts)

- `KEY` (extern) ⟶ `NVS.KEY` (Speicherschlüssel)
- `NumericValue_PHYS.rPhys` ⟶ `NVS.VALUE` (zu speichernder Wert)
- `NVS.VALUEO` ⟶ `Q_NumericValue_PHYS.rPhys` (weitergeleiteter Wert)
- `stObj` (extern) ⟶ `Q_NumericValue_PHYS.stObj` (Konfiguration für die Ausgabe)
- `NVS.VALUEO` ⟶ `VALUEO` (externe Ausgabe)

Zusammenfassung

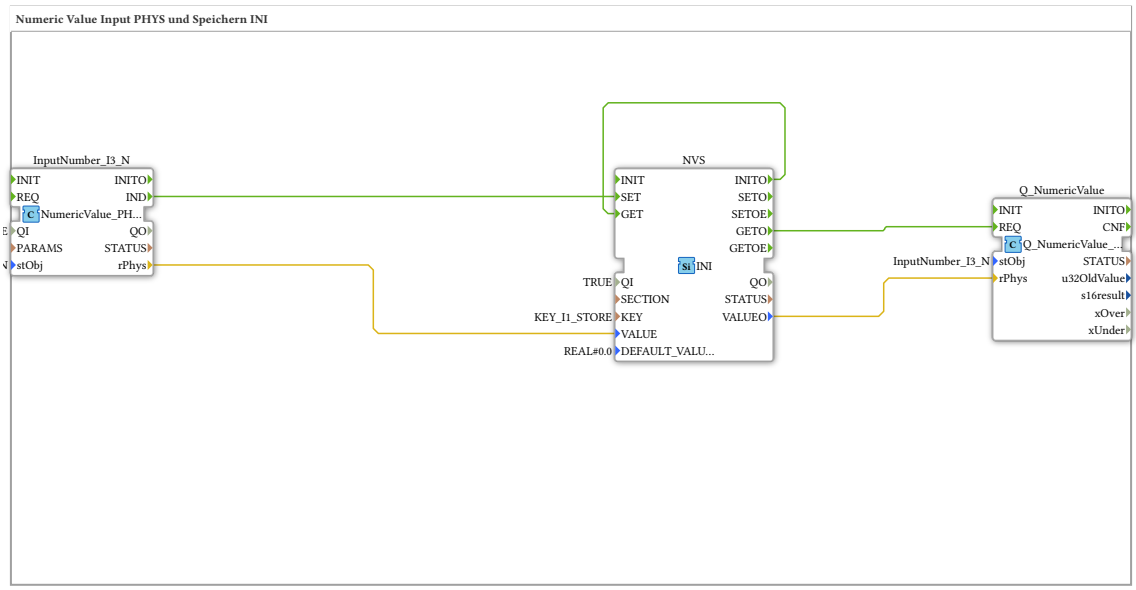
Die Übung `Uebung_012f` zeigt, wie ein physikalischer Messwert (über den `NumericValue_PHYS`-Baustein) eingelesen, persistent in einem nichtflüchtigen Speicher (`NVS`) abgelegt und über einen Qualifier-Baustein ausgegeben wird. Der Schlüssel für die Speicherung und die Objektkonfiguration werden von außen übergeben, was eine flexible Wiederverwendung ermöglicht. Die ereignisgesteuerte Kopplung sorgt für eine korrekte Ablaufreihenfolge von Lesen, Schreiben und Ausgeben. Diese Struktur eignet sich besonders für Anwendungen, bei denen Werte nach dem Einschalten aus einem Speicher wiederhergestellt werden müssen.

2.2.1.165 Uebung_012f_sub

Hinweis

Diese Übung existiert nicht mehr als eigene Übung. Der Baustein wurde in die Bibliothek verschoben und heißt jetzt NumericValue_PHYS_TO_NVS. Beschreibung, Schnittstelle und Netzwerk stehen dort.

2.2.1.166 Uebung_012g: Numeric Value Input PHYS und Speichern INI



• — ○ • — ○ • — ○ •

Einleitung

Diese Übung veranschaulicht die Verwendung eines physischen numerischen Eingangs (`NumericValue_PHYS`) in Verbindung mit einer persistenten Speicherung über das INI-Dateiformat. Ziel ist es, einen eingegebenen numerischen Wert (z. B. von einem Sensor oder einer Benutzereingabe) einmalig zu speichern und bei Bedarf wieder auszulesen. Die Übung vermittelt grundlegende Konzepte der Ereignissteuerung, Datenflussverkettung und nichtflüchtigen Datenspeicherung in 4diac.

Die verwendeten Bausteine stammen aus den Bibliotheken `isobus::UT` und `eclipse4diac::storage`.

Verwendete Funktionsbausteine (FBs)

`InputNumber_I3`

- **Typ**: `isobus::UT::io::NumericValue::NumericValue_PHYS`
- **Parameter**:
  - `QI` = `TRUE`
  - `stObj` = `InputNumber_I3`

- **Funktionsweise**: Dieser Baustein repräsentiert einen physischen numerischen Eingang (z. B. ein Analog- oder Digitaleingang). Bei einer Wertänderung am Eingang wird ein Ereignis `IND` ausgelöst und der aktuelle Wert als Gleitkommazahl am Ausgang `rPhys` bereitgestellt.

`NVS`

- **Typ**: `eclipse4diac::storage::INI`
- **Parameter**:
  - `QI` = `TRUE`
  - `KEY` = `KEY_I1_STORE`
  - `DEFAULT_VALUE` = `REAL#0.0`
- **Funktionsweise**: Der Baustein realisiert eine nichtflüchtige Speicherung mittels INI-Datei. Mit dem Schlüssel `KEY_I1_STORE` kann ein Wert gespeichert und abgerufen werden.
  - Beim Ereignis `INIT` sendet er `INITO` und führt anschließend automatisch `GET` aus.
  - Bei `SET` wird der an `VALUE` anliegende Datenwert persistent gespeichert.
  - Bei `GET` wird der gespeicherte Wert am Ausgang `VALUEO` ausgegeben (falls kein Wert gespeichert ist, wird `DEFAULT_VALUE` verwendet).

`Q_NumericValue`

- **Typ**: `isobus::UT::Q::Q_NumericValue_PHYS`
- **Parameter**:
  - `stObj` = `InputNumber_I3`
- **Funktionsweise**: Dieser Baustein stellt einen physischen Ausgang für numerische Werte dar. Durch die Ereignissanforderung `REQ` wird der an `rPhys` anliegende Wert an den physischen Ausgang weitergegeben.

Programmablauf und Verbindungen

1. **Initialisierung** Beim Start der Subapplikation wird der Baustein `NVS` mit `INIT` initialisiert und sendet `INITO`. Dieses Ereignis löst unmittelbar einen `GET`-Befehl aus, um den zuletzt gespeicherten Wert zu laden. Der ausgelesene Wert gelangt über die Datenverbindung `NVS.VALUEO → Q_NumericValue.rPhys` an den Ausgangsbaustein.
2. **Eingabe eines neuen Wertes** Sobald der Eingangsbaustein `InputNumber_I3` einen neuen numerischen Wert erfasst (Ereignis `IND`), wird dieser Wert über die Datenleitung `InputNumber_I3.rPhys → NVS.VALUE` an den Speicherbaustein übertragen. Gleichzeitig wird durch `IND` das Ereignis `SET` am Baustein `NVS` ausgelöst, wodurch der neue Wert persistent gespeichert wird.
3. **Ausgabe des gespeicherten Wertes** Nach dem Speichervorgang (ausgelöst durch `SET`) führt `NVS` intern einen `GET` durch. Das Ereignis `GETO` wird erzeugt und an `Q_NumericValue.REQ` weitergeleitet. Der zuvor gespeicherte Wert (nun von `NVS.VALUEO` bereitgestellt) wird an den Ausgangsbaustein übergeben und dort physisch ausgegeben.

Zusammenfassung

Die Übung demonstriert eine vollständige Signalverarbeitungskette:

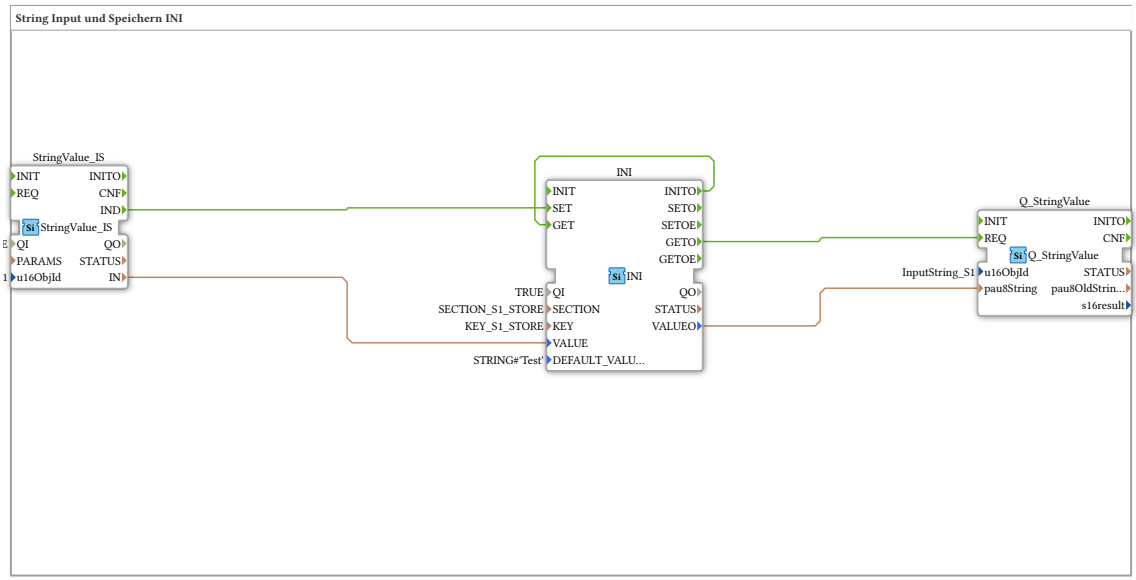
- Erfassen eines numerischen Werts über einen physischen Eingang,
- persistentes Speichern in einer INI-Datei,
- erneutes Ausgeben des gespeicherten Werts an einen physischen Ausgang.

Durch die Ereignis- und Datenverbindungen wird sichergestellt, dass nach dem Einschalten automatisch der letzte gespeicherte Wert angezeigt wird und neue Eingabewerte sofort gespeichert und ausgegeben werden. Dieses Muster eignet sich für Anwendungen wie das Speichern von Sollwerten oder Konfigurationsparametern mit einfacher persistenter Datenhaltung.

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

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

2.2.1.167 Uebung_012j: String Input und Speichern INI



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert die Verarbeitung und Speicherung eines String-Werts mittels eines **INI-Funktionsbausteins**. Ziel ist es, einen von einem Feldbusobjekt (ISOBUS) eingelesenen String in einer INI-Datenstruktur persistent zu speichern und anschließend wieder abzurufen. Die Konfiguration verwendet vordefinierte Konstanten für den Speicherabschnitt (`SECTION_S1_STORE`), den Schlüssel (`KEY_S1_STORE`) sowie die Objekt-ID des Eingabe-Strings (`InputString_S1`).

Verwendete Funktionsbausteine (FBs)

FB: `StringValue_IS`

- **Typ**: `isobus::UT::io::StringValue::StringValue_IS`
- **Parameter**:
  - `QI` = `TRUE` (aktiv)
  - `u16ObjId` = `InputString_S1` (Objekt-ID des ISOBUS-String-Objekts)
- **Ereignisse**:
  - Ereignisausgang: `IND` (wird ausgelöst, wenn ein neuer Stringwert empfangen wird)
- **Daten**:
  - Datenausgang: `IN` (der gelesene String-Wert)
- **Funktionsweise**: Liest bei Aktivierung den Stringwert von der spezifizierten ISOBUS-Objekt-ID (`InputString_S1`) und gibt diesen über den Ausgang `IN` sowie ein Ereignis `IND` aus.

FB: `INI`

- **Typ**: `eclipse4diac::storage::INI`
- **Parameter**:
  - `QI` = `TRUE` (aktiv)
  - `SECTION` = `SECTION_S1_STORE` (Abschnitt in der INI-Datei)
  - `KEY` = `KEY_S1_STORE` (Schlüssel innerhalb des Abschnitts)
  - `DEFAULT_VALUE` = `STRING#'Test'` (Standardwert, falls noch kein Eintrag existiert)
- **Ereignisse**:
  - Ereigniseingänge: `SET` (speichert den anliegenden Wert), `GET` (liest den gespeicherten Wert)
  - Ereignisausgänge: `INITO` (nach erfolgreicher Initialisierung), `GETO` (nach erfolgreichem Lesevorgang)
- **Daten**:
  - Dateneingang: `VALUE` (der zu speichernde String)
  - Datenausgang: `VALUEO` (der ausgelesene String)
- **Funktionsweise**: Der FB verwaltet einen persistenten String-Wert im INI-Format. Beim Ereignis `SET` wird der anliegende `VALUE` unter dem angegebenen Schlüssel und Abschnitt gespeichert. Beim Ereignis

GET wird der gespeicherte Wert auf VALUEO ausgegeben und das Ereignis GETO gesendet. Bei Initialisierung (INITO) wird automatisch ein GET ausgeführt.

FB: Q_StringValue

- **Typ**: isobus::UT::Q::Q_StringValue
- **Parameter**:
  - u16ObjId = InputString_S1 (Objekt-ID – wird hier nicht direkt verwendet, aber als Kontext)
- **Ereignisse**:
  - Ereigniseingang: REQ (Anforderung zur Ausgabe)
- **Daten**:
  - Dateneingang: pau8String (der auszugebende String)
- **Funktionsweise**: Nimmt einen String entgegen und stellt ihn auf dem ISOBUS-Objekt mit der angegebenen ID zur Verfügung (z. B. zur Anzeige auf einem Terminal).

Programmablauf und Verbindungen

Der Programmablauf gliedert sich in zwei Phasen: **Initialisierung** und **zyklische Verarbeitung**.

Ereignisverbindungen

1. **Initialisierung**: Der FB INI erzeugt nach erfolgreicher Initialisierung das Ereignis INITO. Dieses wird direkt mit dem GET-Eingang von INI verbunden. Dadurch wird unmittelbar nach dem Start der gespeicherte Wert gelesen.
2. **Lesen des gespeicherten Werts**: Nach dem Lesevorgang gibt INI das Ereignis GETO aus. Dieses triggert den REQ-Eingang von Q_StringValue, sodass der ausgelesene String an das ISOBUS-Objekt übergeben wird.
3. **Speichern eines neuen Werts**: Wenn StringValue_IS einen neuen String vom ISOBUS-Objekt empfängt, sendet es das Ereignis IND. Dieses ist mit dem SET-Eingang von INI verbunden, sodass der neue Wert gespeichert wird.

Datenverbindungen

- Der Ausgang IN von StringValue_IS wird mit dem Dateneingang VALUE von INI verbunden – der gelesene String wird zum Speichern weitergegeben.
- Der Ausgang VALUEO von INI wird mit dem Dateneingang pau8String von Q_StringValue verbunden – der ausgelesene String wird für die Ausgabe bereitgestellt.

Ablaufdiagramm (vereinfacht)

1. **Start**: INI initialisiert → INITO → GET → liest gespeicherten Wert → GETO → Q_StringValue.REQ → Ausgabe des gespeicherten Strings.
2. **Neuer Eingang**: StringValue_IS empfängt neuen String → IND → INI.SET → speichert den neuen Wert.
3. Nach einem erneuten GET (z. B. durch zyklischen Trigger) wird der aktuell gespeicherte Wert ausgegeben.

Zusammenfassung

Die Übung Uebung_012j vermittelt den Umgang mit:

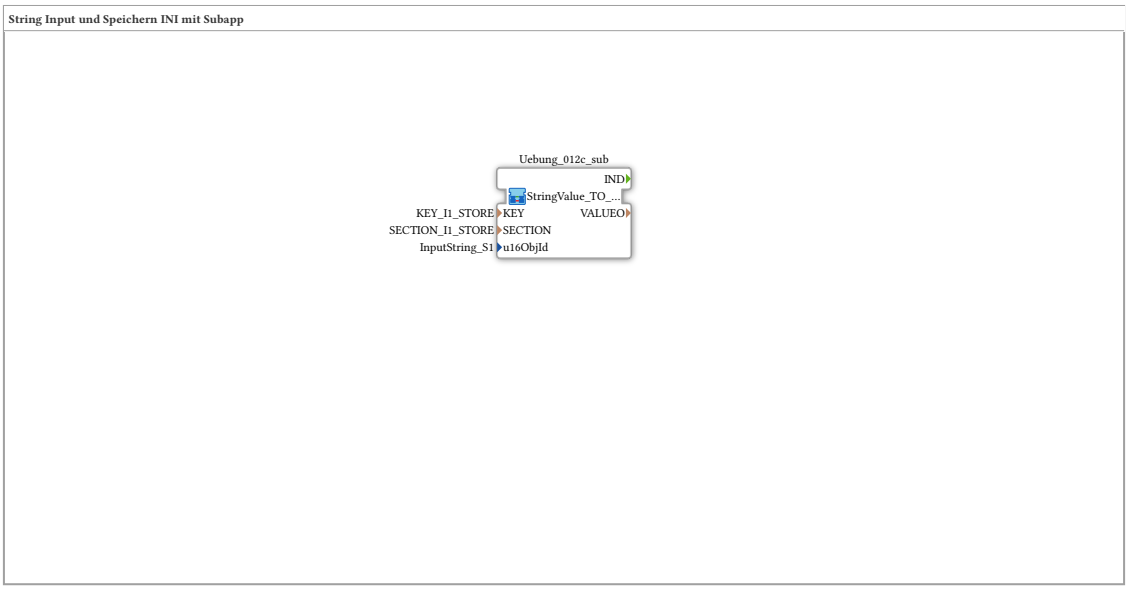
- Einlesen eines String-Werts von einem ISOBUS-Objekt (StringValue_IS)
- Persistenter Speicherung des Werts mittels INI-Funktionsbaustein (Abschnitt, Schlüssel, Standardwert)
- Rückgabe des gespeicherten Werts an ein ISOBUS-Objekt (Q_StringValue)

Durch die Verwendung von Konstanten (SECTION_S1_STORE, KEY_S1_STORE, InputString_S1) wird eine klare Trennung zwischen Konfiguration und Logik erreicht. Der Ablauf zeigt eine typische Initialisierungs- und Update-Strategie für dezentrale Steuerungssysteme mit Speicherbedarf.

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

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

2.2.1.168 Uebung_012k: String Input und Speichern INI mit Subapp



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung demonstriert das Einlesen eines String-Wertes von einer Eingabequelle (z. B. einem Touch-Eingabefeld) und das dauerhafte Speichern dieses Wertes in einer INI-Datei. Die gesamte Funktionalität ist in einer Subapp gekapselt, die als wiederverwendbarer Baustein zur Verfügung steht. Der gespeicherte Wert steht nach jedem Lese- oder Speichervorgang als Ausgang zur Verfügung und wird gleichzeitig an eine Ausgabekomponente (z. B. Display) übermittelt.

Verwendete Funktionsbausteine (FBs)

Sub-Baustein: `Uebung_012k_sub`

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **`StringValue_IS`**: `isobus::UT::io::StringValue::StringValue_IS`
    - Parameter: `QI` = `TRUE`
    - Beschreibung: Dieser FB liest den aktuellen String-Wert von einer über `u16ObjId` spezifizierten Eingabequelle. Wird ein neuer Wert erkannt, erzeugt er ein Ereignis am Ausgang `IND`.
  - **`INI`**: `eclipse4diac::storage::INI`
    - Parameter: `QI` = `TRUE`, `DEFAULT_VALUE` = `STRING#''`
    - Beschreibung: Der FB speichert einen String-Wert unter einem bestimmten Schlüssel (`KEY`) und Bereich (`SECTION`) in einer INI-Datei. Er unterstützt die Ereignisse `SET` (Speichern), `GET` (Laden) und `INIT` (Initialisierung).
  - **`Q_StringValue`**: `isobus::UT::Q::Q_StringValue`
    - Parameter: keine (Parameter werden über Datenverbindungen gesetzt)
    - Beschreibung: Dieser FB konvertiert den von `INI` gelieferten Binär-String in ein Ausgabeformat und sendet ihn an die durch `u16ObjId` identifizierte Ausgabestelle (z. B. Display).
- **Funktionsweise**:
  1. **Initialisierung**: Beim Start wird das `INIT`-Ereignis des `INI`-FBs ausgelöst. Dieses triggert intern ein `GET`, sodass der zuletzt gespeicherte Wert aus der INI-Datei geladen wird.
  2. **Speichern eines neuen Wertes**: Wenn der `StringValue_IS`-FB einen neuen Eingabewert erkennt, sendet er ein Ereignis an den `SET`-Eingang von `INI`. Daraufhin speichert `INI` den aktuellen Wert unter dem vorgegebenen `KEY` und `SECTION` ab und gibt das Ereignis `SETO` aus.
  3. **Lesen und Ausgeben**: Sobald `INI` einen `GET`-Vorgang abschliesst (sowohl nach INIT als auch nach jedem SET – hier jedoch nur nach INIT realisiert), wird über `GETO` das Ereignis an den `REQ`-Eingang von `Q_StringValue` und gleichzeitig an den Ausgang `IND` der SubApp weitergeleitet. Der geladene String wird über `VALUEO` direkt nach aussen gegeben und von `Q_StringValue` an die Ausgabestelle gesendet.

Programmablauf und Verbindungen

Der oberste Baustein `Uebung_012k` besitzt keine eigene Schnittstelle (leere `SubAppInterfaceList`). Er instanziiert den Sub-Baustein `Uebung_012k_sub` und übergibt diesem drei konstante Parameter:

- `KEY` = `KEY_I1_STORE`
- `SECTION` = `SECTION_I1_STORE`
- `u16ObjId` = `InputString_S1`

Diese Konstanten sind in den referenzierten Bibliotheken definiert und legen fest, unter welchem Schlüssel und Bereich der String in der INI-Datei abgelegt wird, sowie welche Eingabe-/Ausgabekomponente verwendet wird.

Verbindungen im Sub-Baustein:

- Ereignisfluss:
  - `StringValue_IS.IND` ⟶ `INI.SET`
  - `INI.SETO` ⟶ `IND` (Ausgang der SubApp)
  - `INI.GETO` ⟶ `Q_StringValue.REQ` und `IND`
  - `INI.INITO` ⟶ `INI.GET`
- Datenfluss:
  - `StringValue_IS.IN` ⟶ `INI.VALUE`
  - `u16ObjId` ⟶ `StringValue_IS.u16ObjId` und `Q_StringValue.u16ObjId`
  - `KEY` ⟶ `INI.KEY`
  - `SECTION` ⟶ `INI.SECTION`
  - `INI.VALUEO` ⟶ `Q_StringValue.pau8String` und `VALUEO` (Ausgang der SubApp)

Beachte: Nach einem Speichervorgang (`SET`) wird aktuell kein erneutes `GET` ausgelöst. Daher wird der neu gespeicherte Wert erst bei der nächsten Initialisierung oder einem manuellen `GET` gelesen und an die Ausgabe weitergegeben. In der vorliegenden Konfiguration dient dies dem einmaligen Speichern eines Startwertes, der beim Neustart wiederhergestellt wird.

Zusammenfassung

Die Übung zeigt den Umgang mit String-Eingaben, der persistierenden Speicherung in INI-Dateien sowie der gekapselten Wiederverwendung von Funktionalität durch eine Subapp. Der `INI`-FB übernimmt das Speichern und Laden, während `StringValue_IS` und `Q_StringValue` die Kommunikation mit der Hardware (Eingabe/Ausgabe) realisieren. Die Subapp `Uebung_012k_sub` kann in übergeordneten Projekten einfach per Parameterkonfiguration an unterschiedliche Objekt-IDs und Speicherorte angepasst werden.

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

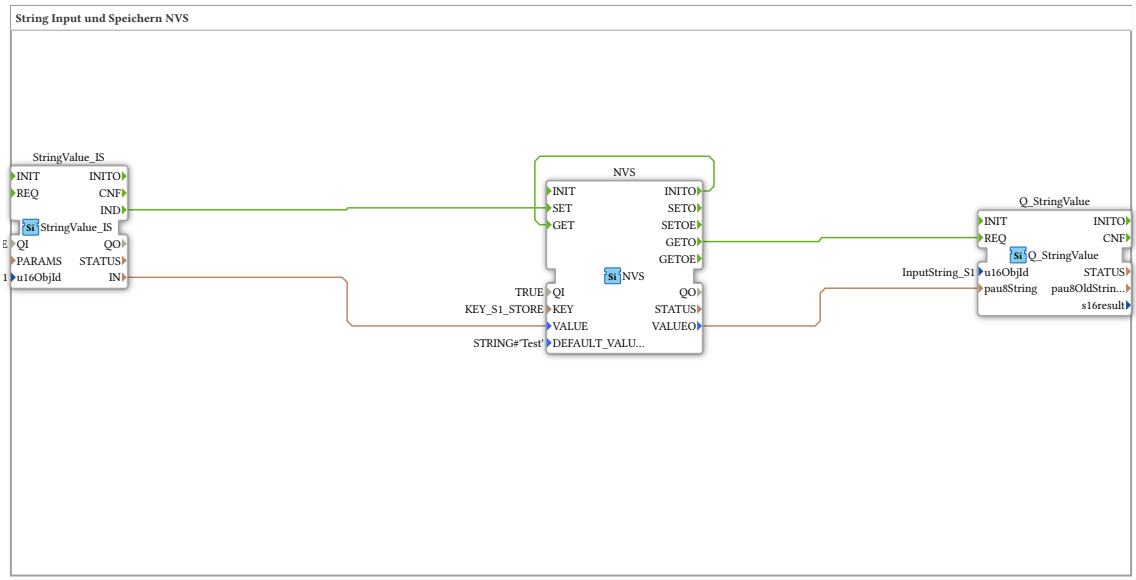
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.2.1.169 Uebung_012k_sub

Hinweis

Diese Übung existiert nicht mehr als eigene Übung. Der Baustein wurde in die Bibliothek verschoben und heißt jetzt StringValue_TO_INI. Beschreibung, Schnittstelle und Netzwerk stehen dort.

2.2.1.170 Uebung_012l: String Input und Speichern NVS



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert das Einlesen eines Strings von einer ISOBUS-Variablen (InputString_S1) und das Speichern dieses Wertes im nichtflüchtigen Speicher (NVS) eines ESP32. Beim Starten der Applikation wird der zuletzt gespeicherte String automatisch aus dem NVS geladen und an die entsprechende ISOBUS-Variable zurückgeschrieben. Somit bleibt der Wert auch nach einem Neustart erhalten.

Verwendete Funktionsbausteine (FBs)

StringValue_IS

- **Typ**: `isobus::UT::io::StringValue::StringValue_IS`
- **Parameter**:
  - `QI` = `TRUE`
  - `u16ObjId` = `InputString_S1`
- **Funktionsweise**: Der Baustein empfängt den aktuellen Stringwert der ISOBUS-Objektvariablen `InputString_S1`. Wann immer sich dieser Wert ändert (z. B. durch eine Eingabe am Terminal), wird am Ereignisausgang `IND` ein Signal erzeugt und der neue String am Datenausgang `IN` bereitgestellt.

NVS

- **Typ**: `logiBUS::storage::esp32_nvs::NVS`
- **Parameter**:
  - `QI` = `TRUE`
  - `KEY` = `KEY_S1_STORE`
  - `DEFAULT_VALUE` = `STRING#'Test'`
- **Funktionsweise**: Der Baustein verwaltet den nichtflüchtigen Speicher auf dem ESP32.
  - Bei einem Ereignis an `SET` speichert er den an `VALUE` anliegenden String unter dem Schlüssel `KEY_S1_STORE` ab.
  - Bei einem Ereignis an `GET` lädt er den gespeicherten String und stellt ihn am Datenausgang `VALUEO` bereit.
  - Nach der Initialisierung (`INITO`) löst er automatisch einen `GET` aus.

Q_StringValue

- **Typ**: `isobus::UT::Q::Q_StringValue`
- **Parameter**:
  - `u16ObjId` = `InputString_S1`

- **Funktionsweise**: Dieser Baustein schreibt („qualifiziert“) einen empfangenen String zurück in die ISOBUS-Variable `InputString_S1`. Ein Ereignis an `REQ` übernimmt den Wert von `pau8String` und aktualisiert die ISOBUS-Variable.

Programmablauf und Verbindungen

Initialisierung (Start)

1. Nach dem Start des Controllers erhält der `NVS`-Baustein das Ereignis `INITO`.
2. Dieses Ereignis wird intern mit dem Ausgang `GET` verbunden (im XML als `Connection Source="NVS.INITO" Destination="NVS.GET" …` sichtbar). Somit wird sofort ein Lesevorgang ausgelöst.
3. Der `NVS`-Baustein lädt den unter `KEY_S1_STORE` gespeicherten String (oder den Default-Wert `"Test"`, falls noch kein Wert gespeichert wurde) und gibt ihn am Ausgang `VALUEO` aus.
4. Gleichzeitig erzeugt der `NVS` am Ausgang `GETO` ein Ereignis, das mit dem `REQ`-Eingang von `Q_StringValue` verbunden ist.
5. `Q_StringValue` übernimmt den String von `NVS.VALUEO` (via `pau8String`) und schreibt ihn in die ISOBUS-Variable `InputString_S1`.

Änderung des Strings (z. B. durch Terminaleingabe)

1. Sobald der Wert von `InputString_S1` von außen geändert wird (z. B. über ein Bedienterminal), erzeugt `StringValue_IS` ein Ereignis an `IND`.
2. Dieses Ereignis ist mit dem `SET`-Eingang des `NVS`-Bausteins verbunden.
3. `NVS` speichert den aktuellen String (von `StringValue_IS.IN` über die Datenverbindung an `NVS.VALUE`) unter dem Schlüssel `KEY_S1_STORE`.
4. **Hinweis:** Nach dem Speichern wird **nicht** automatisch der `Q_StringValue` aktualisiert. Die Rückschreibung in die ISOBUS-Variable erfolgt nur beim Start. Das ist beabsichtigt, da der Wert ja bereits im Terminal sichtbar ist.

Datenverbindungen im Überblick

- **Ereignisse**:
  - `NVS.INITO` ⟶ `NVS.GET` (initialer Lesevorgang)
  - `NVS.GETO` ⟶ `Q_StringValue.REQ` (Ausgabe des geladenen Strings)
  - `StringValue_IS.IND` ⟶ `NVS.SET` (Speicherung bei Änderung)
- **Daten**:
  - `NVS.VALUEO` ⟶ `Q_StringValue.pau8String` (zu ladender String)
  - `StringValue_IS.IN` ⟶ `NVS.VALUE` (zu speichernder String)

Wichtige Konstanten

- **`KEY_S1_STORE`**: Der NVS-Schlüssel, unter dem der String gespeichert wird.
- **`InputString_S1`**: Die ID der ISOBUS-Stringvariablen, die als Quelle und Ziel dient.

Lernziele

- Verständnis der nichtflüchtigen Speicherung (NVS) auf ESP32-Systemen.
- Umgang mit ISOBUS-Stringvariablen in 4diac.
- Ereignisgesteuerte Abläufe: Initialisierung und reaktive Speicherung.
- Verwendung von vordefinierten Konstanten (Schlüssel, Objekt-IDs).

Benötigte Vorkenntnisse

- Grundlegende Bedienung der 4diac-IDE.
- Grundlagen der ISOBUS-Kommunikation (Objekt-IDs, Werte lesen/schreiben).
- Einfaches Verständnis von ereignisgesteuerten Systemen.

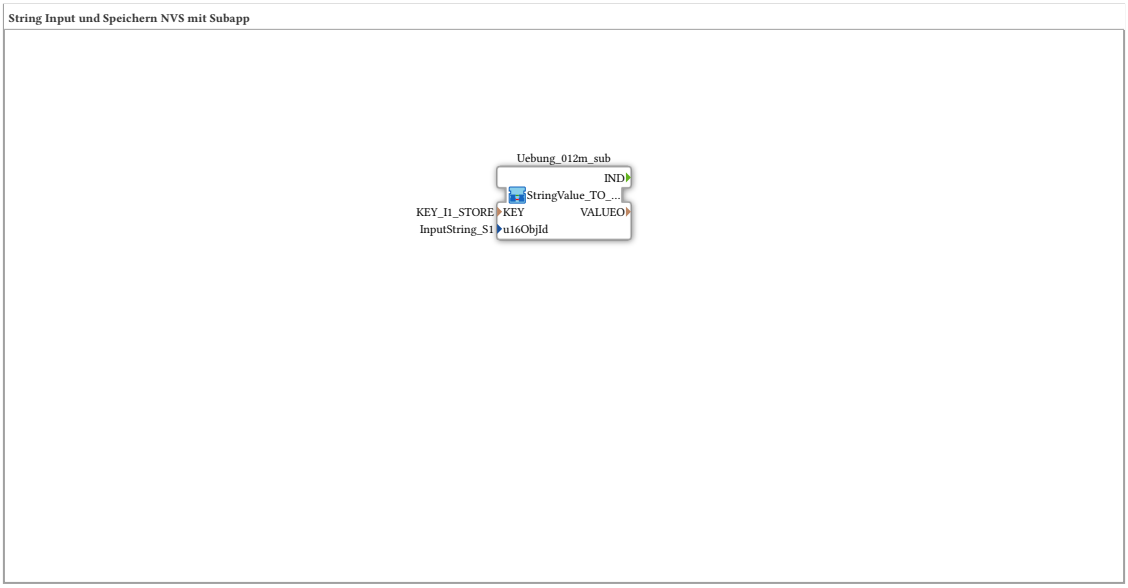
Zusammenfassung

Die Übung `Uebung_012l` zeigt, wie ein ISOBUS-String in den NVS-Speicher des ESP32 geschrieben und beim Systemstart wieder ausgelesen wird. Der Wert bleibt dauerhaft erhalten, auch nach einem Neustart oder Spannungsausfall. Die Applikation besteht aus drei Funktionsbausteinen, die über Ereignis- und Datenverbindungen zusammenarbeiten und demonstriert einen typischen Anwendungsfall für persistenten Variablenzugriff in der Landtechnik-Automatisierung.

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

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de
- 🌐 ESP32 & ESP32-S3 DevKit auf ms-muc-docs.de

2.2.1.171 Uebung_012m: String Input und Speichern NVS mit Subapp



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung demonstriert die Verarbeitung eines Zeichenketten-Inputs (String) aus einer isobus-Quelle und das Speichern sowie Auslesen des Wertes in/aus dem nichtflüchtigen Speicher (NVS) eines ESP32. Die gesamte Logik ist in einer gekapselten SubApp organisiert, die die Wiederverwendbarkeit und Modularität fördert. Ziel ist es, eine robuste Speicherung von Konfigurationsdaten oder Benutzereingaben zu realisieren, die auch nach einem Neustart erhalten bleibt.

Verwendete Funktionsbausteine (FBs)

Sub-Baustein: Uebung_012m_sub

- **Typ**: `SubAppType` (eigene SubApp)
- **Verwendete interne FBs**:
  - **StringValue_IS**: `isobus::UT::io::StringValue::StringValue_IS`
    - **Parameter**: `QI = TRUE`
    - **Ereignisausgang/-eingang**:
      - Eingang: (kein expliziter Ereigniseingang im XML, wird über `IND` von darüber liegender Logik gesteuert – hier nicht direkt sichtbar)
      - Ausgang: `IND` – signalisiert, dass der Wert von der isobus-Quelle gelesen wurde
    - **Datenausgang/-eingang**:
      - Eingang: `u16ObjId` (über SubApp-Parameter verbunden) – Identifikation des isobus-Objekts
      - Ausgang: `IN` – der gelesene String-Wert
  - **NVS**: `logiBUS::storage::esp32_nvs::NVS`
    - **Parameter**: `QI = TRUE`, `DEFAULT_VALUE = STRING#''`
    - **Ereigniseingänge/-ausgänge**:
      - Eingänge: `SET` (schreiben), `GET` (lesen), `INIT` (Initialisierung)
      - Ausgänge: `SETO` (Schreiben bestätigt), `GETO` (Lesen bestätigt), `INITO` (Initialisierung abgeschlossen)
    - **Dateneingänge/-ausgänge**:
      - Eingänge: `KEY` (Schlüssel für NVS-Eintrag), `VALUE` (zu speichernder Wert)
      - Ausgänge: `VALUEO` (ausgelesener Wert)

  - **Q_StringValue**: `isobus::UT::Q::Q_StringValue`
    - **Parameter**: keine sichtbaren in der XML
    - **Ereigniseingang/-ausgang**:
      - Eingang: `REQ` – Anforderung zur Ausgabe des Wertes auf den isobus-Bus
      - Ausgang: (nicht im XML gelistet, dient der Bus-Kommunikation)
    - **Dateneingänge/-ausgänge**:
      - Eingang: `u16ObjId` (Objekt-ID für isobus), `pau8String` (Zeiger auf String-Daten)
      - Ausgang: keiner sichtbar (sendet über isobus)

- **Funktionsweise**: Die SubApp `Uebung_012m_sub` kapselt den gesamten Ablauf: Einlesen eines Strings über `StringValue_IS`, Speichern im NVS über den FB `NVS`, und optionales Senden des gespeicherten Wertes über den isobus-Bus mit `Q_StringValue`. Die Parameter `KEY` und `u16ObjId` werden von außen übergeben und bestimmen den NVS-Speicherplatz bzw. die isobus-Objekt-ID.

  Ablauf:

  1. Nach der Initialisierung (`NVS.INITO`) wird automatisch ein Lesevorgang (`NVS.GET`) ausgelöst. Der ausgelesene Wert liegt dann an `NVS.VALUEO`.
  2. Bei einem Event von außen (nicht gezeigt, aber logisch) wird `StringValue_IS` aktiviert. Dessen Ausgang `IND` triggert `NVS.SET`, wodurch der aktuelle String im NVS gespeichert wird.
  3. Nach erfolgreichem Schreiben (`NVS.SETO`) oder Lesen (`NVS.GETO`) wird ein Event an den Ausgang `IND` der SubApp gesendet.
  4. Gleichzeitig wird bei jedem Lesevorgang (`NVS.GETO`) der ausgelesene Wert über `Q_StringValue.REQ` an den isobus-Bus gesendet, sodass andere Geräte den aktuellen Wert abfragen können.

Programmablauf und Verbindungen

Die Haupt-SubApp `Uebung_012m` instanziiert `Uebung_012m_sub` und versorgt diese mit zwei Parametern:

- `KEY` = `KEY_I1_STORE` (ein konstanter Schlüssel für den NVS-Eintrag)
- `u16ObjId` = `InputString_S1` (die isobus-Objekt-ID, von der der Eingabestring gelesen wird)

Die Verbindungen innerhalb von `Uebung_012m_sub` sind wie folgt (aus der XML-Datei):

- **Ereignisflüsse**:
  - `StringValue_IS.IND` ⟶ `NVS.SET` (nach Einlesen eines neuen Strings wird dieser sofort gespeichert)
  - `NVS.SETO` ⟶ SubApp-Eventausgang `IND` (Signal nach erfolgreichem Schreiben)
  - `NVS.GETO` ⟶ `Q_StringValue.REQ` sowie ⟶ SubApp-Eventausgang `IND` (nach Lesen wird der Wert auf den Bus gegeben und Signal ausgegeben)
  - `NVS.INITO` ⟶ `NVS.GET` (direkt nach Initialisierung wird der gespeicherte Wert gelesen)
- **Datenflüsse**:
  - `StringValue_IS.IN` ⟶ `NVS.VALUE` (der eingelesene String wird dem NVS zum Speichern übergeben)
  - `NVS.VALUEO` ⟶ `Q_StringValue.pau8String` (der ausgelesene String wird zum Senden vorbereitet)
  - `NVS.VALUEO` ⟶ SubApp-Datenausgang `VALUEO` (zur Weitergabe an darüberliegende Schichten)
  - `KEY` (extern) ⟶ `NVS.KEY`
  - `u16ObjId` (extern) ⟶ `StringValue_IS.u16ObjId` und `Q_StringValue.u16ObjId`

Hinweise:

- Die Übung setzt voraus, dass die isobus-Bibliothek und der NVS-Treiber für den ESP32 in der Laufzeitumgebung vorhanden sind.
- Der Parameter `DEFAULT_VALUE = STRING#''` definiert einen leeren String als Standard, falls im NVS noch kein Wert hinterlegt ist.
- Die Verbindungen sind als sichtbar markiert (außer einige mit `Visible = false` – diese dienen lediglich der internen Verschaltung und sind in der grafischen Darstellung der IDE standardmäßig ausgeblendet).

Zusammenfassung

In dieser Übung wird der Umgang mit String-Eingaben, nichtflüchtigem Speicher (NVS) und isobus-Kommunikation in einer gekapselten SubApp trainiert. Die Lernziele umfassen:

- Verständnis für die Initialisierung und Nutzung des NVS-FBs.
- Ereignisgesteuerte Verkettung von Lese-, Schreib- und Kommunikationsvorgängen.
- Parametrisierung von SubApps zur Wiederverwendung.
- Integration von isobus-Objekten zur Datenübertragung.

Schwierigkeitsgrad: Mittel **Benötigte Vorkenntnisse**: Grundlagen der 4diac-IDE, Ereignis- und Datenverbindungen, einfache FBs. **Start der Übung**: Laden Sie die Haupt-SubApp `Uebung_012m` in ein Projekt und verbinden Sie die Eingänge `KEY` und `u16ObjId` mit entsprechenden Konstanten oder Variablen. Die Ausgabe `VALUE0` kann z. B. an eine Anzeige angeschlossen werden.

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

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de
- 🌐 ESP32 & ESP32-S3 DevKit auf ms-muc-docs.de

2.2.1.172 Uebung_012m_sub

Hinweis

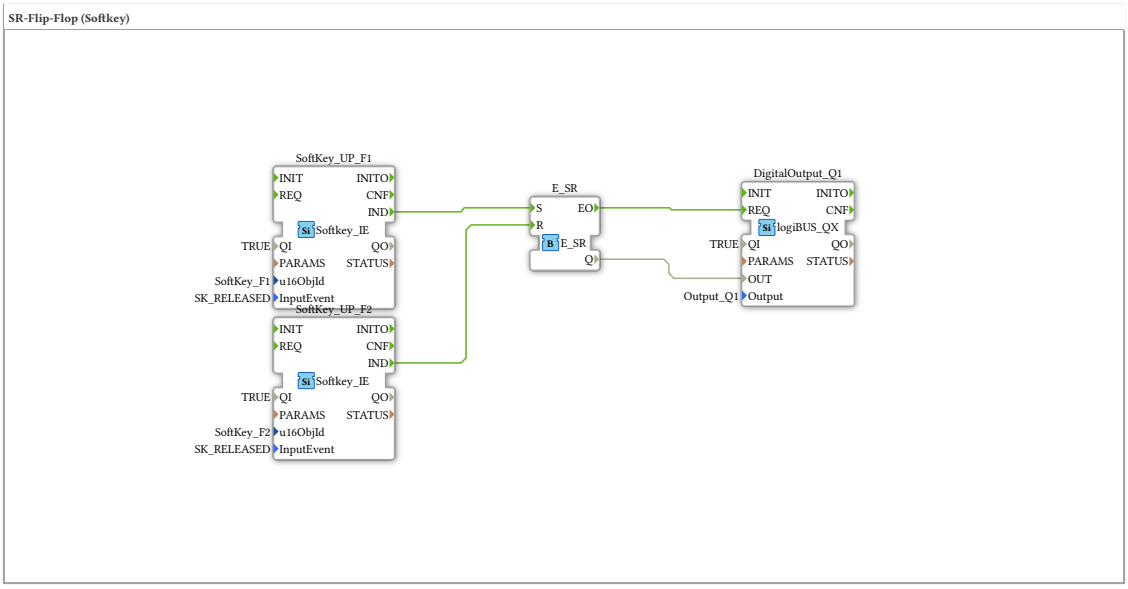
Diese Übung existiert nicht mehr als eigene Übung. Der Baustein wurde in die Bibliothek verschoben und heißt jetzt StringValue_TO_NVS. Beschreibung, Schnittstelle und Netzwerk stehen dort.

2.2.1.173 Uebung_013: SR-Flip-Flop (Softkey)



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_013`. Hier wird eine Speicherfunktion realisiert, die vollständig über das ISOBUS-Terminal bedient wird.

🎧 Podcast



- Die drei Timer der DIN EN 61131-3 entschlüsselt – TP, TON & TOF präzise erklärt
- DIN EN 61131-3 vs. 61499-1: Dein Wegweiser durch die Normen der Industrieautomatisierung
- DIN EN 61131-3: Das Herz der Land- und Baumaschinen-Mechatronik und der Sprung in die Zukunft mit Ob
- FB_TOF und E_TOF: Verzögerungstimer in IEC 61131-3 und 61499
- IEC 61499 vs. 61131: Brauchen wir einen neuen Standard für IIoT? Analyse einer hitzigen Debatte um Verteilte Intelligenz

---

Ziel der Übung

Realisierung einer Ein/Aus-Schaltung mit getrennten virtuellen Tasten.

Beschreibung und Komponenten

Die Subapplikation `Uebung_013.SUB` nutzt zwei Softkeys zur Steuerung eines SR-Flip-Flops.

Funktionsbausteine (FBs)

- **`SoftKey_UP_F1`**: Triggert beim Loslassen (Release) den Set-Eingang (`S`).
- **`SoftKey_UP_F2`**: Triggert beim Loslassen (Release) den Reset-Eingang (`R`).
- **`E_SR`**: Der Speicherbaustein.
- **`DigitalOutput_Q1`**: Der Hardware-Ausgang.

Funktionsweise

- Ein Klick auf **F1** schaltet die Funktion ein.
- Ein Klick auf **F2** schaltet die Funktion aus.

Die Verwendung von `SK_RELEASED` sorgt für ein stabiles Bediengefühl am Touchscreen. Da zwei getrennte Tasten verwendet werden, ist der Zustand der Anlage für den Bediener immer eindeutig steuerbar.

Anwendungsbeispiel

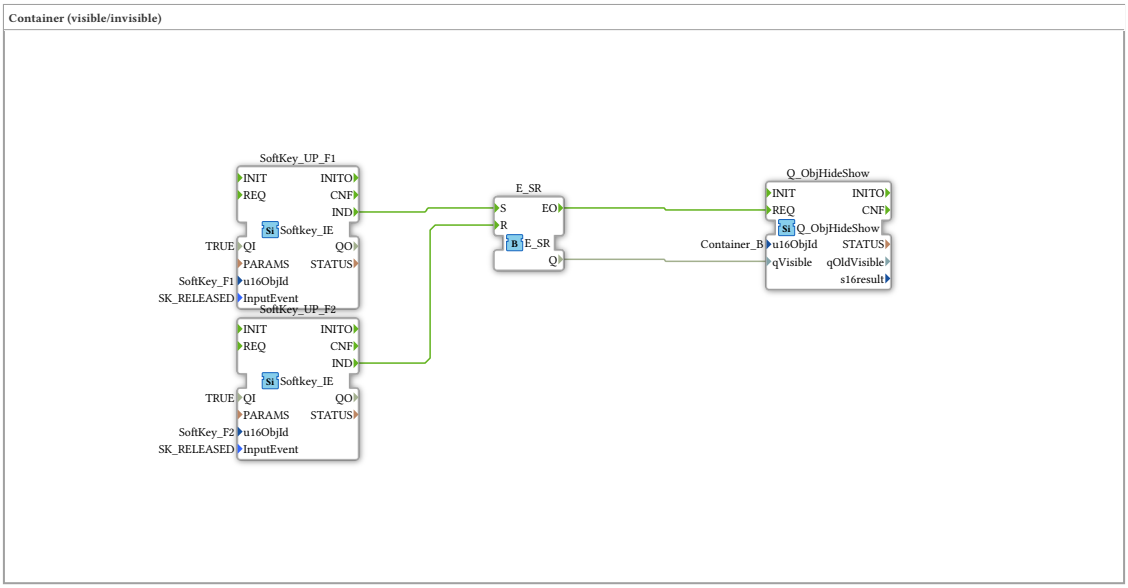
Zuschalten eines Anbaugeräts: Am Terminal gibt es zwei deutliche Symbole: Ein grüner Haken (`F1`) für „System Aktiv“ und ein rotes Kreuz (`F2`) für „System Deaktiviert“. Der Speicher in der Steuerung sorgt dafür, dass die gewählte Betriebsart erhalten bleibt, bis die jeweils andere Taste gedrückt wird.

2.2.1.174 Uebung_014: Container (visible/invisible)



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_014`. Hier wird gezeigt, wie man die Benutzeroberfläche des ISOBUS-Terminals dynamisch verändert, indem ganze Gruppen von Objekten (Container) sichtbar oder unsichtbar geschaltet werden.

🎧 Podcast



- 4diac IDE: Dein Open-Source-Werkzeugkasten für verteilte Industrieautomatisierung nach IEC 61499
- 4diac IDE: Wie der IEC 61499 Standard die Industrieautomatisierung revolutioniert
- Eclipse 4diac FORTE: IEC 61499 verstehen – Der LEGO®-Baukasten für Ihre Industrie 4.0 Steuerung
- Eclipse 4diac: Open-Source-Automatisierung für Industrie und Forschung nach IEC 61499
- IEC 61499: Die Zukunft der industriellen Automatisierung und verteilten Systeme

---

Ziel der Übung

Verwendung des Bausteins `Q_ObjHideShow` zur Steuerung der Sichtbarkeit von ISOBUS-Objekten. Dies ermöglicht es, kontextsensitive Oberflächen zu erstellen, die nur die Informationen anzeigen, die im aktuellen Betriebszustand relevant sind.

Beschreibung und Komponenten

Die Subapplikation `Uebung_014.SUB` nutzt zwei Softkeys, um einen Speicher zu setzen oder zu löschen, dessen Zustand die Sichtbarkeit eines Containers steuert.

Funktionsbausteine (FBs)

- **`SoftKey_UP_F1` & `F2`**: Eingabe am Terminal (Ein/Aus).
- **`E_SR`**: Der Speicher für den Sichtbarkeits-Status.
- **`Q_ObjHideShow`**: Der ISOBUS-Ausgangsbaustein. Er steuert die Eigenschaft „Sichtbarkeit“ des Objekts mit der ID `Container_B`.

Funktionsweise

1. Ein Klick auf **F1** setzt den Speicher auf `TRUE`.
2. Ein Klick auf **F2** setzt den Speicher auf `FALSE`.
3. Das jeweilige Ereignis (`EO`) triggert den `REQ`-Eingang von `Q_ObjHideShow`.
4. Der Baustein übermittelt den Zustand von `qVisible` an das Terminal.
5. Alle Objekte, die sich im ISOBUS-Pool innerhalb des `Container_B` befinden, erscheinen oder verschwinden nun zeitgleich auf dem Bildschirm.

Anwendungsbeispiel

Optionale Ausstattungsmerkmale: Eine Maschine kann mit oder ohne Wiegeeinrichtung bestellt werden. In der Software ist die Wiege-Anzeige in einem Container gruppiert. Je nach Konfiguration (oder Knopfdruck) wird dieser gesamte Bereich ein- oder ausgeblendet, sodass der Bediener nicht durch inaktive Felder abgelenkt wird.

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

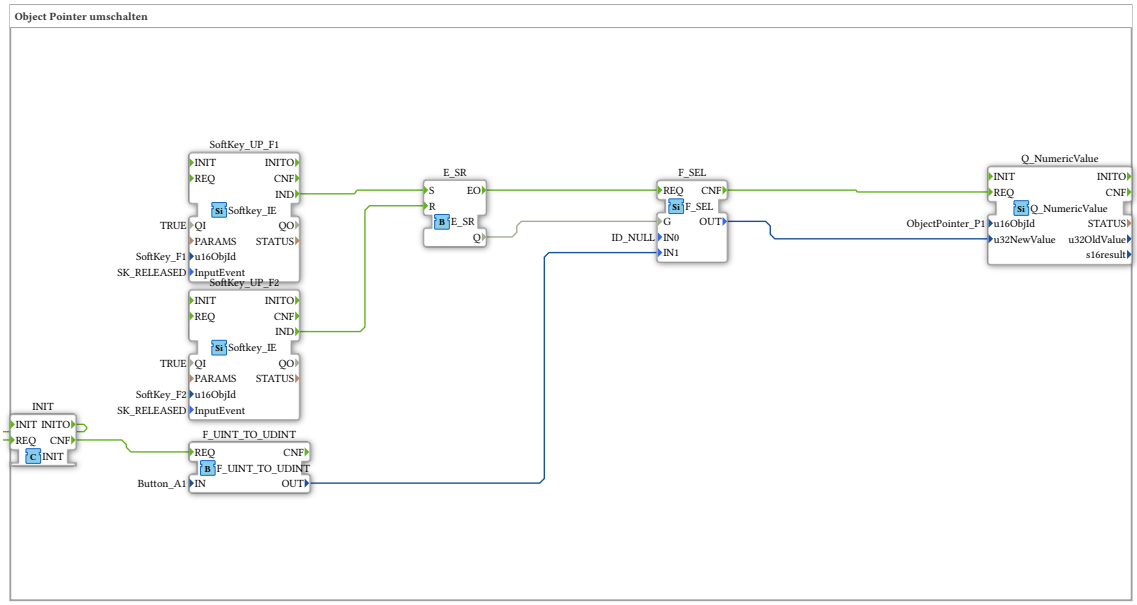
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.2.1.175 Uebung_015: Object Pointer umschalten



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_015`. Hier wird eine fortgeschrittene ISOBUS-Technik demonstriert: Das Umschalten von Object Pointern, um Bildschirminhalte dynamisch auszutauschen.

🎧 Podcast



- Als Landtechnik-Spezialist durch die Hölle: Wie Lanz-Wery Krieg, Besatzung und Hyperinflation überlebte – Einblicke in Original-Geschäftsberichte 1915-1922
- Hannes' Turbo-Mais: Wie ein Landwirt mit Hackschnitzel-Kreislauf und Turmtrockner 15.000 Tonnen Körnermais verarbeitet
- JBC Lötspitzen C470 vs. C245 vs. C210 vs. C115: Welche Spitze ist der Allrounder und wann brauchst du den Nano-Spezialisten?
- Schlüter 1500 Spezial: Turbo-Giftigkeit, 40 Jahre und die Seele eines Kraftprotzes

---

Ziel der Übung

Erlernen der Verwendung von `Object Pointer` Objekten. Ein Pointer ist ein Platzhalter auf dem Bildschirm, dem zur Laufzeit die ID eines anderen Objekts zugewiesen werden kann. Dies ist effizienter als das Ausblenden vieler Einzelobjekte.

Beschreibung und Komponenten

In `Uebung_015.SUB` wird ein Object Pointer (`ObjectPointer_P1`) zwischen einer Schaltfläche (`Button_A1`) und einem leeren Zustand (`ID_NULL`) umgeschaltet.

Funktionsbausteine (FBs)

- **`SoftKey_UP_F1` & `F2`**: Steuern die Auswahl.
- **`F_SEL`**: Ein Auswahl-Baustein (Selection). Je nach Eingang `G` (vom Speicher `E_SR`) gibt er entweder den Wert `ID_NULL` (0) oder die Objekt-ID von `Button_A1` aus.
- **`Q_NumericValue`**: Wird hier zweckentfremdet, um die ID an den Pointer zu senden (da ein Pointer-Update technisch das Senden einer neuen ID an die Pointer-Objekt-ID ist).

Funktionsweise

1. Nutzer drückt **F1** ➡️ Speicher wird `TRUE` ➡️ `F_SEL` schaltet `Button_A1` durch.
2. Die ID von `Button_A1` wird an `ObjectPointer_P1` gesendet.
3. Auf dem Bildschirm erscheint an der Position des Pointers plötzlich die Schaltfläche `A1`.
4. Nutzer drückt **F2** ➡️ ID `0` wird gesendet ➡️ Die Stelle auf dem Bildschirm wird wieder leer.

Anwendungsbeispiel

Kontextsensitive Buttons: Ein zentraler Platz auf dem Terminal soll je nach Arbeitsmodus unterschiedliche Funktionen anzeigen (z.B. im Modus „Transport“ ein Straßensymbol, im Modus „Feld“ ein

Pflugsymbol). Anstatt zwei Buttons übereinander zu legen und zu verstecken, wird ein Pointer genutzt, der je nach Modus auf das eine oder andere Bild-Objekt verweist.

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

- 🌐 Interaktiver JBC-Lötspitzen Guide & Infografik auf ms-muc-docs.de

2.2.1.176 Uebung_015a: Object Pointer umschalten – 3-fach



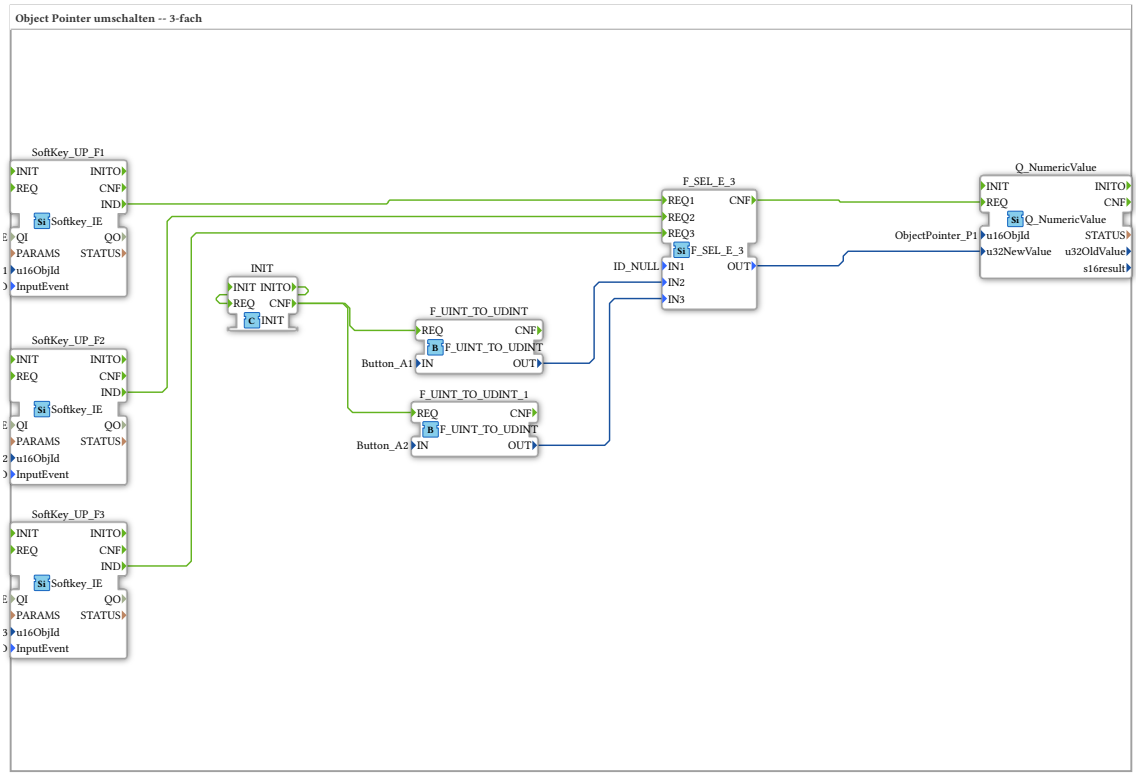
Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_015a`.

Übersicht

Diese Übung erweitert das Pointer-Konzept aus Übung 015 auf drei Zustände unter Verwendung des Bausteins `F_SEL_E_3`. Über drei Softkeys (`F1`, `F2`, `F3`) kann der Nutzer entscheiden, was an einer bestimmten Stelle auf dem Bildschirm angezeigt wird:

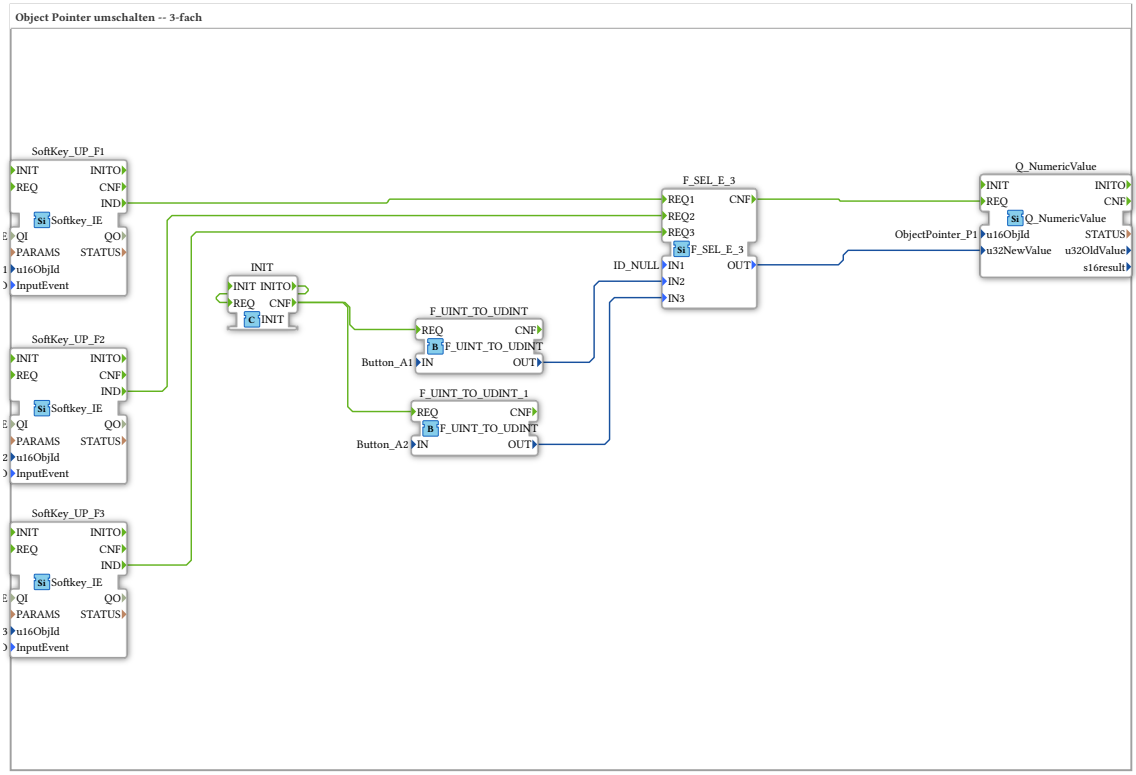
1. Nichts (`ID_NULL`)
2. Schaltfläche `Button_A1`
3. Schaltfläche `Button_A2`

Dies demonstriert die Flexibilität von Pointern bei der Erstellung von dynamischen Menüstrukturen oder umschaltbaren Info-Bereichen.



2.2.1.177 Uebung_015b: Object Pointer umschalten – 3-fach

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_015b (Object Pointer umschalten – 3-fach).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Object Pointer umschalten – 3-fach**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_015b.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **SoftKey_UP_F1**: Instanz des Typs isobus::UT::io::Softkey::Softkey_IE.
  - Parameter QI = TRUE
  - Parameter u16ObjId = SoftKey_F1
  - Parameter InputEvent = SK_RELEASED
- **SoftKey_UP_F2**: Instanz des Typs isobus::UT::io::Softkey::Softkey_IE.
  - Parameter QI = TRUE
  - Parameter u16ObjId = SoftKey_F2
  - Parameter InputEvent = SK_RELEASED
- **F_UINT_TO_UDINT**: Instanz des Typs iec61131::conversion::F_UINT_TO_UDINT.
  - Parameter IN = Button_A1
- **F_SEL_E_3**: Instanz des Typs eclipse4diac::utils::selection::F_SEL_E_3.
  - Parameter IN1 = ID_NULL
- **Q_NumericValue**: Instanz des Typs isobus::UT::Q::Q_NumericValue.
  - Parameter u16ObjId = ObjectPointer_P1
- **SoftKey_UP_F3**: Instanz des Typs isobus::UT::io::Softkey::Softkey_IE.
  - Parameter QI = TRUE
  - Parameter u16ObjId = SoftKey_F3
  - Parameter InputEvent = SK_RELEASED
- **F_UINT_TO_UDINT_1**: Instanz des Typs iec61131::conversion::F_UINT_TO_UDINT.
  - Parameter IN = Button_A2
- **INIT**: Instanz des Typs iec61131::booleanOperators::INIT.

Verbindungen und Schnittstellen

Ereignisverbindungen:

- F_SEL_E_3.CNF -> Q_NumericValue.REQ
- SoftKey_UP_F1.IND -> F_SEL_E_3.REQ1
- SoftKey_UP_F2.IND -> F_SEL_E_3.REQ2
- SoftKey_UP_F3.IND -> F_SEL_E_3.REQ3
- INIT.INITO -> INIT.REQ
- INIT.CNF -> F_UINT_TO_UDINT.REQ
- INIT.CNF -> F_UINT_TO_UDINT_1.REQ

Datenverbindungen:

- F_UINT_TO_UDINT.OUT -> F_SEL_E_3.IN2
- F_SEL_E_3.OUT -> Q_NumericValue.u32NewValue
- F_UINT_TO_UDINT_1.OUT -> F_SEL_E_3.IN3

Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

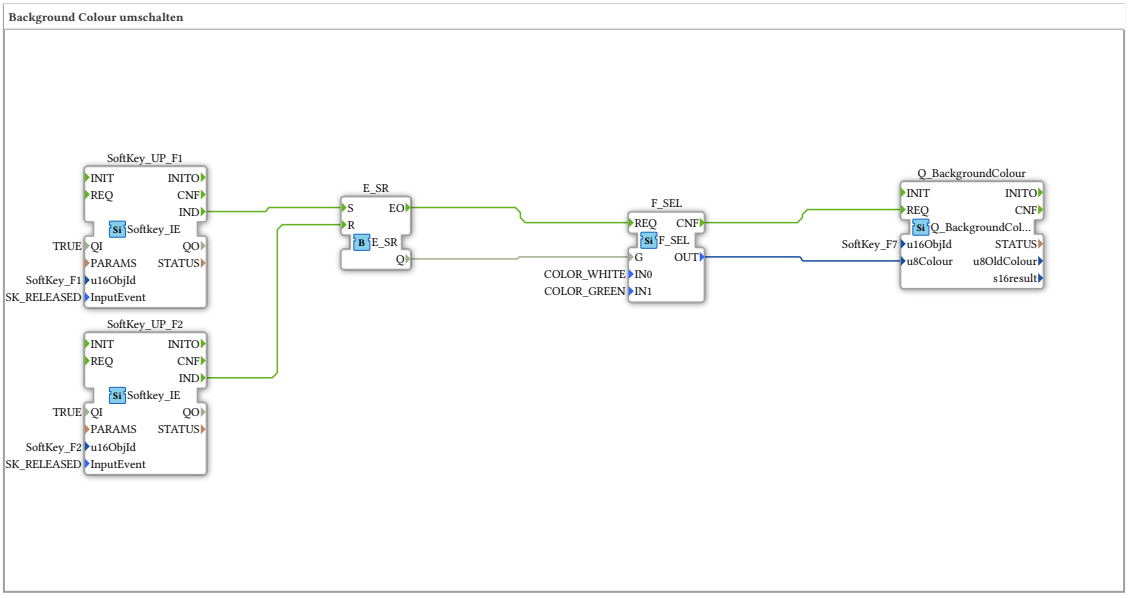
Die Übung Uebung_015b demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.2.1.178 Uebung_016: Background Colour umschalten



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_016`. Hier wird gezeigt, wie man die Hintergrundfarbe von Objekten (z.B. Softkeys) zur Laufzeit ändert, um Zustände zu visualisieren.

🎧 Podcast



- ESP32-S3-DevKitC-1 Doku-Analyse: Das Speicher-Monster (32MB Flash/16MB PSRAM) und die Macht der Dual-USB-Ports

---

Ziel der Übung

Verwendung des Bausteins `Q_BackgroundColour`. Dies ist eine Alternative zum Farbumschlag in Sub-Applikationen (wie in Übung 010c) und erlaubt die explizite Wahl von Farben aus der ISOBUS-Palette.

Beschreibung und Komponenten

Die Subapplikation `Uebung_016.SUB` schaltet die Farbe des Softkeys `F7` basierend auf der Auswahl über `F1` und `F2` um.

Funktionsbausteine (FBs)

- **`F_SEL`**: Wählt zwischen zwei Farb-Konstanten aus.
- **`Q_BackgroundColour`**: Der Ausgangsbaustein. Er setzt die Hintergrundfarbe für das Objekt `SoftKey_F7`.

Funktionsweise

- Wird der Speicher durch **F1** gesetzt, liefert `F_SEL` den Wert `COLOR_GREEN`.
- Wird er durch **F2** gelöscht, liefert `F_SEL` den Wert `COLOR_WHITE`.
- Das Ergebnis wird an `Q_BackgroundColour` gesendet, welches das entsprechende ISOBUS-Kommando („Change Background Colour“) an das Terminal absetzt.

Der Softkey `F7` (der in dieser Übung keine eigene Logik hat, sondern nur als Anzeige dient) wechselt nun zwischen Grün und Weiß.

Anwendungsbeispiel

Status-Ampel: Ein Sensor überwacht einen Füllstand. Ist alles im grünen Bereich, leuchtet eine Anzeige am Terminal grün. Erreicht der Stand eine kritische Marke, schaltet die Anzeige auf Gelb oder Rot um, um den Bediener visuell zu warnen.

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

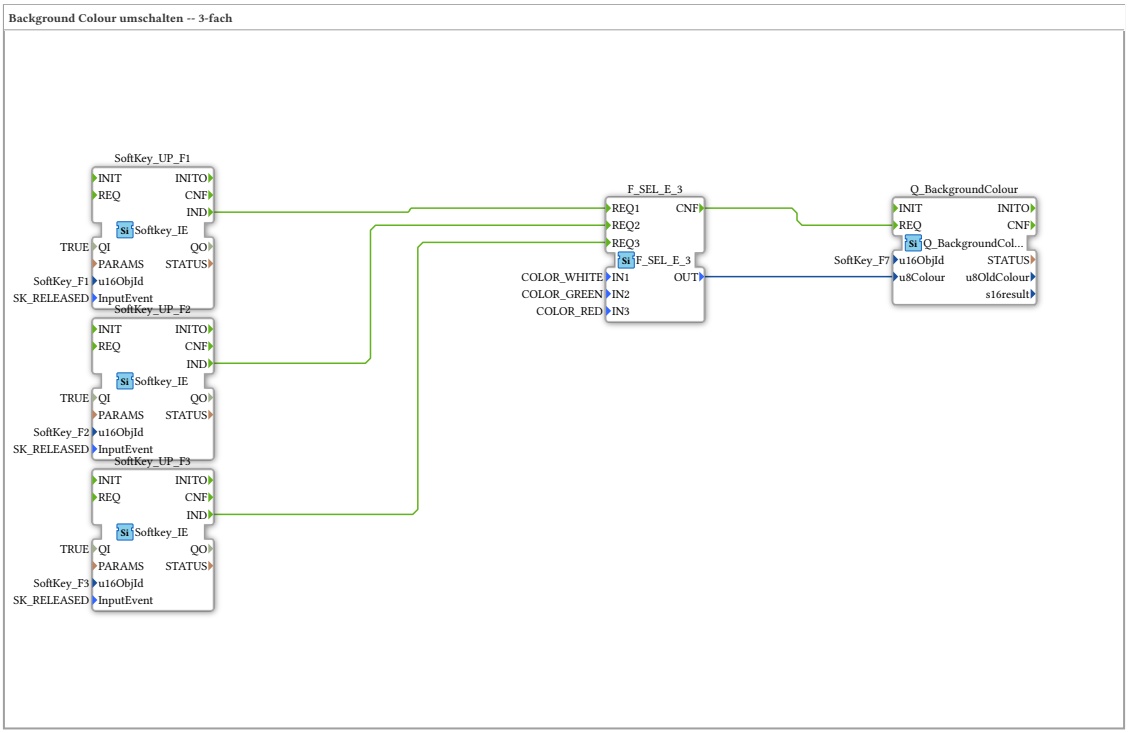
- 🌐 ESP32 & ESP32-S3 DevKit auf ms-muc-docs.de

2.2.1.179 Uebung_016a: Background Colour umschalten – 3-fach



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_016a`.

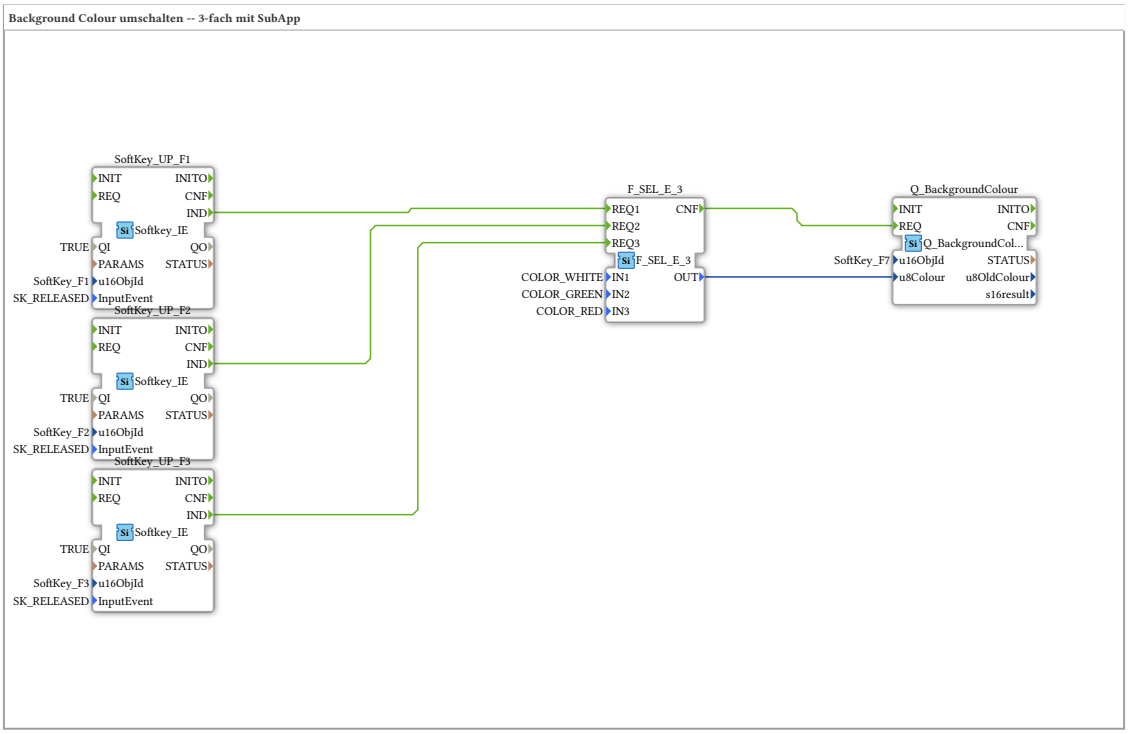
Übersicht



Diese Übung erweitert das Farb-Umschaltkonzept auf drei Farben unter Verwendung von `F_SEL_E_3`. Über drei Softkeys kann der Hintergrund eines Objekts (`SoftKey_F7`) direkt auf **Weiß**, **Grün** oder **Rot** gesetzt werden. Dies ist die Grundlage für komplexere Ampel-Logiken oder differenzierte Statusmeldungen am Universal Terminal.

2.2.1.180 Uebung_016b: Background Colour umschalten – 3-fach mit SubApp

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_016b (Background Colour umschalten – 3-fach mit SubApp).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Background Colour umschalten – 3-fach mit SubApp**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_016b.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **SoftKey_UP_F1**: Instanz des Typs isobus::UT::io::Softkey::Softkey_IE.
  - Parameter QI = TRUE
  - Parameter u16ObjId = SoftKey_F1
  - Parameter InputEvent = SK_RELEASED
- **SoftKey_UP_F2**: Instanz des Typs isobus::UT::io::Softkey::Softkey_IE.
  - Parameter QI = TRUE
  - Parameter u16ObjId = SoftKey_F2
  - Parameter InputEvent = SK_RELEASED
- **SoftKey_UP_F3**: Instanz des Typs isobus::UT::io::Softkey::Softkey_IE.
  - Parameter QI = TRUE
  - Parameter u16ObjId = SoftKey_F3
  - Parameter InputEvent = SK_RELEASED
- **F_SEL_E_3**: Instanz des Typs eclipse4diac::utils::selection::F_SEL_E_3.
  - Parameter IN1 = COLOR_WHITE
  - Parameter IN2 = COLOR_GREEN
  - Parameter IN3 = COLOR_RED
- **Q_BackgroundColour**: Instanz des Typs isobus::UT::Q::Q_BackgroundColour.
  - Parameter u16ObjId = SoftKey_F7

Verbindungen und Schnittstellen

Ereignisverbindungen:

- F_SEL_E_3.CNF -> Q_BackgroundColour.REQ
- SoftKey_UP_F1.IND -> F_SEL_E_3.REQ1
- SoftKey_UP_F2.IND -> F_SEL_E_3.REQ2
- SoftKey_UP_F3.IND -> F_SEL_E_3.REQ3

Datenverbindungen:

- F_SEL_E_3.OUT -> Q_BackgroundColour.u8Colour

Programmablauf und Funktionsweise

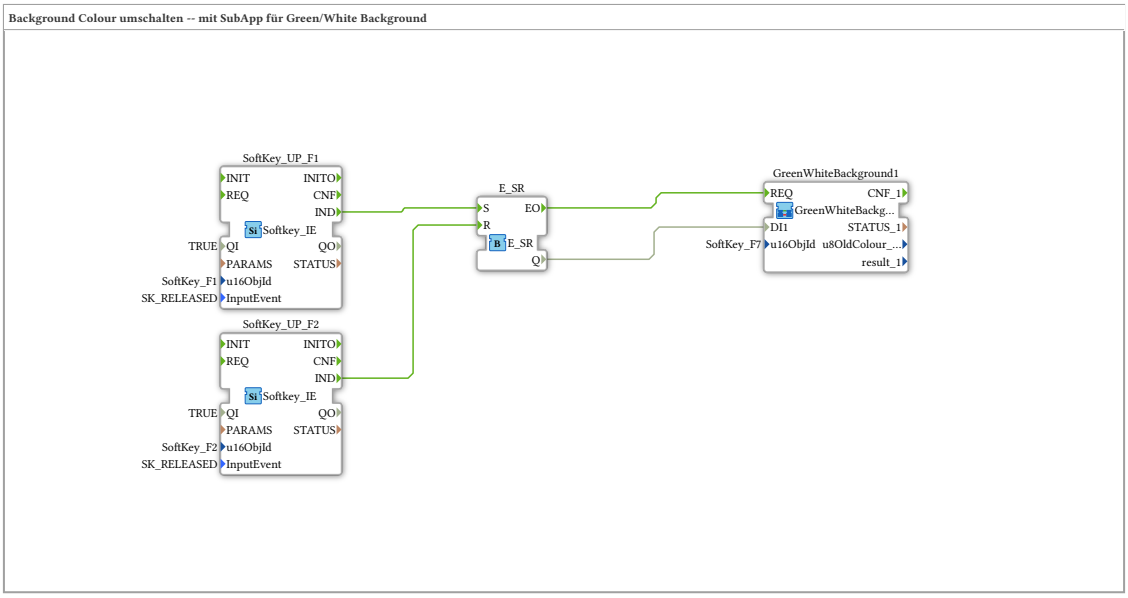
1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_016b demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.2.1.181 Uebung_016c: Background Colour umschalten – mit SubApp für Green/White Background

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_016c (Background Colour umschalten – mit SubApp für Green/White Background).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Background Colour umschalten – mit SubApp für Green/White Background**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_016c.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **SoftKey_UP_F1**: Instanz des Typs isobus::UT::io::Softkey::Softkey_IE.
  - Parameter QI = TRUE
  - Parameter u16ObjId = SoftKey_F1
  - Parameter InputEvent = SK_RELEASED
- **SoftKey_UP_F2**: Instanz des Typs isobus::UT::io::Softkey::Softkey_IE.
  - Parameter QI = TRUE
  - Parameter u16ObjId = SoftKey_F2
  - Parameter InputEvent = SK_RELEASED
- **E_SR**: Instanz des Typs iec61499::events::E_SR.

Verbindungen und Schnittstellen

Ereignisverbindungen:

- SoftKey_UP_F1.IND -> E_SR.S
- SoftKey_UP_F2.IND -> E_SR.R
- E_SR.EO -> GreenWhiteBackground1.REQ

Datenverbindungen:

- E_SR.Q -> GreenWhiteBackground1.DI1

Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

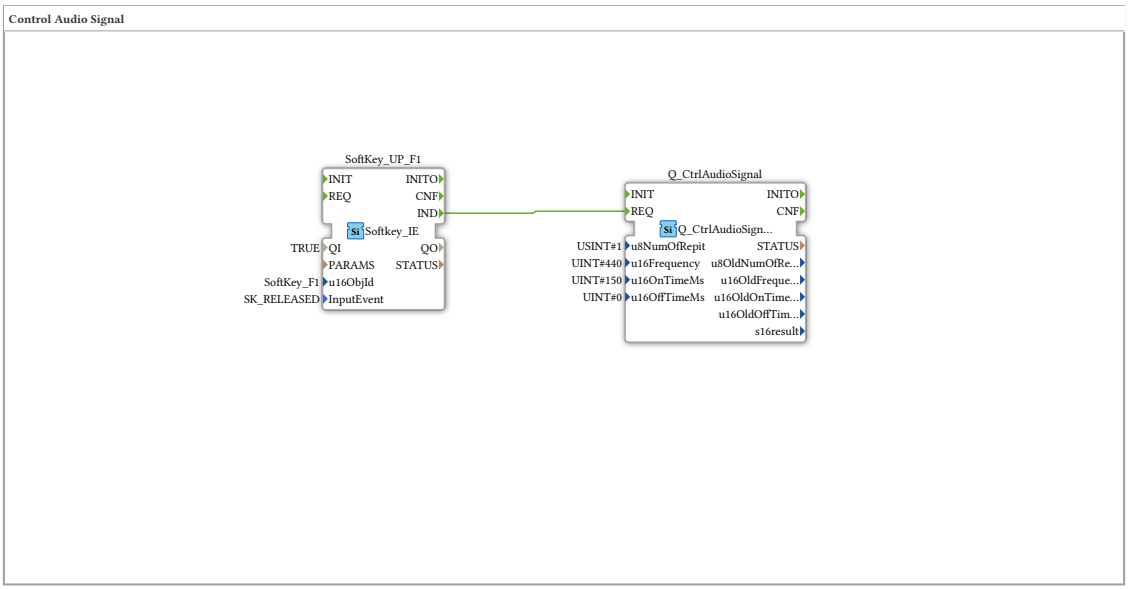
Zusammenfassung

Die Übung Uebung_016c demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.2.1.182 Uebung_017: Control Audio Signal

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_017`. In dieser Übung wird gezeigt, wie man den internen Summer des ISOBUS-Terminals anspricht, um akustische Rückmeldungen zu geben.

Podcast



- „Store Version“ – Dein Schlüssel zur Verwaltung von Objektdatenpools im nichtflüchtigen VT-Speicher (ISO 11783-6)
- ISO 11783-6: Softkeys und das Virtual Terminal verstehen – Dein Schlüssel zur Landmaschinen-Mechatronik
- ISOBUS Skalierung: Wenn der Ackerschlepper-Bildschirm nicht passt – Eine Einführung in ISO 11783-6
- ISOBUS-Balkendiagramm: Das Output Linear Bar Graph Objekt der ISO 11783-6 entschlüsselt
- ISOBUS-Bedienoberflächen: Wenn Tasten und Hauptanzeige unterschiedlich skalieren – ISO 11783-6 entschlüsselt

---

Ziel der Übung

Verwendung des Bausteins `Q_CtrlAudioSignal`. Es wird demonstriert, wie ein Ereignis (hier ein Softkey-Klick) eine Audio-Ausgabe am Terminal mit spezifischer Frequenz und Dauer auslöst.

Beschreibung und Komponenten

Die Subapplikation `Uebung_017.SUB` löst bei Betätigung eines Softkeys ein Tonsignal aus.

Funktionsbausteine (FBs)

- **`SoftKey_UP_F1`**: Der Auslöser.
- **`Q_CtrlAudioSignal`**: Der ISOBUS-Ausgangsbaustein für Audio.
- **Parameter**:
  - `u16Frequency`: Tonhöhe in Hertz (hier 440 Hz = Kammerton A).
  - `u16OnTimeMs`: Dauer des Tons (150 ms).
  - `u8NumOfRepit`: Anzahl der Wiederholungen (1).

Funktionsweise

Die Kette ist rein ereignisbasiert: Ein Klick (und Loslassen) von Softkey **F1** feuert ein `IND`-Event. Dieses geht direkt an den `REQ`-Eingang des Audio-Bausteins. Das Terminal erhält daraufhin den Befehl, einmalig für 150ms mit 440 Hz zu piepsen.

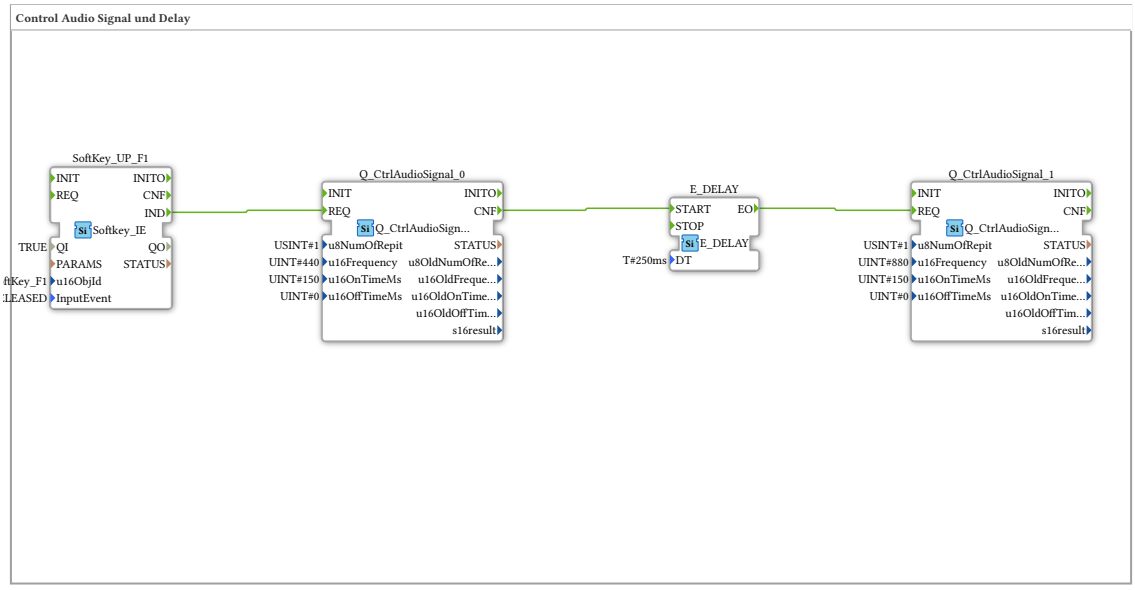
Anwendungsbeispiel

Tastenton-Quittierung: Jeder Tastendruck am Terminal soll durch einen kurzen, dezenten Piepston bestätigt werden. Dies gibt dem Bediener eine akustische Rückmeldung über die erfolgreiche Eingabe, auch wenn er nicht direkt auf den Bildschirm schaut.

2.2.1.183 Uebung_018: Control Audio Signal und Delay

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_018`. Hier wird die Audio-Ausgabe erweitert, um zeitlich versetzte Tonfolgen zu erzeugen.

📺 Video



- Der Katalog von 1863

🎧 Podcast

- Agrar-Revolution 1883: Wie Max Eyth Englands Landwirtschaft modernisierte
- Apfelwein-Allzweckwaffe und Stickstoff-Revolution: Die Landwirtschaft Mittelfrankens 1892 im Zeitungs-Check
- Das Technologie-Panorama von 1863: Lanz & Comp. und die Revolution der deutschen Landwirtschaft durch Import, Innovation und Guano

---

Ziel der Übung

Erlernen der Ereignisverzögerung (`E_DELAY`) zur Erstellung von Sequenzen. Es wird gezeigt, wie man zwei Töne mit unterschiedlichen Frequenzen nacheinander abspielt.

Beschreibung und Komponenten

In `Uebung_018.SUB` werden zwei Audio-Bausteine über ein Zeitglied verkettet.

Funktionsbausteine (FBs)

- **`Q_CtrlAudioSignal_0`**: Erster Ton (440 Hz).

- **`E_DELAY`**: Ein Verzögerungs-Baustein. Er wartet nach einem Ereignis am Eingang `START` die Zeit `DT` ab (hier 250 ms), bevor er das Ereignis am Ausgang `EO` weitergibt.
- **`Q_CtrlAudioSignal_1`**: Zweiter Ton (880 Hz - eine Oktave höher).

Funktionsweise

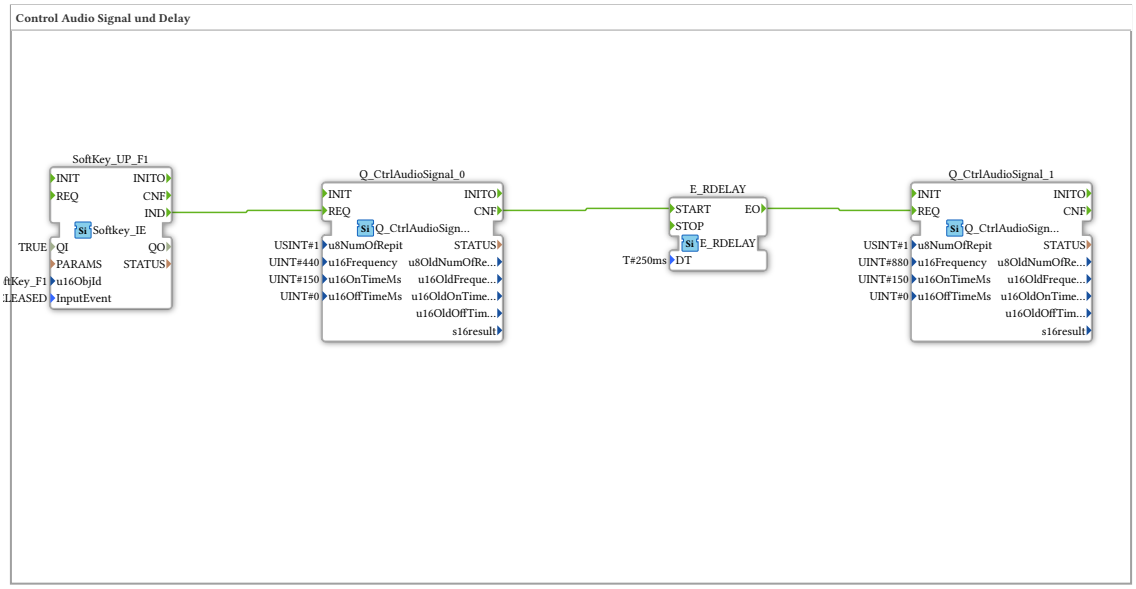
1. Softkey-Klick startet Ton 0.
2. Gleichzeitig mit dem Start des ersten Tons (oder nach dessen Bestätigung `CNF`) wird der Timer `E_DELAY` gestartet.
3. Während der erste Ton klingt (150ms) und die kurze Pause danach, läuft die Zeit ab.
4. Nach 250ms feuert der Timer und startet den zweiten (höheren) Ton.

Das Ergebnis ist ein zweistufiges „Didi“-Signal.

Anwendungsbeispiel

Differenzierte Warnsignale: Ein kurzes „Piep“ ist eine normale Info. Ein „Piep-Piep“ (z.B. tiefer Ton gefolgt von hohem Ton) signalisiert das Ende eines Vorgangs. Ein umgekehrtes Signal (hoch nach tief) könnte eine Fehlermeldung akustisch untermalen.

2.2.1.184 Uebung_018a: Control Audio Signal und Delay



• — ○ • — ○ • — ○ •

Einleitung

Diese Übung demonstriert die Steuerung von Audiosignalen in einer ISOBUS-Universal-Terminal-Umgebung in Kombination mit einer Zeitverzögerung. Ziel ist es, beim Loslassen einer Softkey-Taste eine Sequenz aus zwei unterschiedlichen Tönen abzuspielen, die durch eine kurze Pause getrennt sind. Dies veranschaulicht die Ereignisverarbeitung und die Verwendung von Delay-Bausteinen zur Sequenzierung von Aktionen.

Verwendete Funktionsbausteine (FBs)

In dieser SubApplikation werden verschiedene Funktionsbausteine instanziiert und verschaltet, um die gewünschte Logik zu realisieren.

Sub-Bausteine

SoftKey_UP_F1

- **Typ**: `isobus::UT::io::Softkey::Softkey_IE`
- **Beschreibung**: Dieser Baustein überwacht die Eingabe am Universal Terminal (UT) für eine spezifische Softkey-Taste.

- **Konfiguration**:
  - **Parameter**: `QI` = `TRUE` (Aktiviert den Baustein)
  - **Parameter**: `u16ObjId` = `SoftKey_F1` (Identifikator der Taste F1)
  - **Parameter**: `InputEvent` = `SK_RELEASED` (Reagiert auf das Loslassen der Taste)
- **Ereignisausgang**: `IND` (Signalisiert, dass das Ereignis eingetreten ist)

Q_CtrlAudioSignal_0

- **Typ**: `isobus::UT::Q::Q_CtrlAudioSignal`
- **Beschreibung**: Erzeugt das erste akustische Signal (tieferer Ton).
- **Konfiguration**:
  - **Parameter**: `u8NumOfRepit` = `1` (Einmaliges Abspielen)
  - **Parameter**: `u16Frequency` = `440` (Frequenz in Hz - Kammerton A)
  - **Parameter**: `u16OnTimeMs` = `150` (Dauer des Tons in Millisekunden)
  - **Parameter**: `u16OffTimeMs` = `0`
- **Ereigniseingang**: `REQ` (Startet die Tonausgabe)
- **Ereignisausgang**: `CNF` (Bestätigt die Verarbeitung)

E_RDELAY

- **Typ**: `iec61499::events::E_RDELAY`
- **Beschreibung**: Ein Verzögerungsbaustein, der das Weiterleiten eines Ereignisses um eine definierte Zeit verschiebt.
- **Konfiguration**:
  - **Parameter**: `DT` = `T#250ms` (Verzögerungszeit von 250 Millisekunden)
- **Ereigniseingang**: `START` (Startet den Timer)
- **Ereignisausgang**: `EO` (Feuert nach Ablauf der Zeit)

Q_CtrlAudioSignal_1

- **Typ**: `isobus::UT::Q::Q_CtrlAudioSignal`
- **Beschreibung**: Erzeugt das zweite akustische Signal (höherer Ton).
- **Konfiguration**:
  - **Parameter**: `u8NumOfRepit` = `1` (Einmaliges Abspielen)
  - **Parameter**: `u16Frequency` = `880` (Frequenz in Hz - eine Oktave höher)
  - **Parameter**: `u16OnTimeMs` = `150` (Dauer des Tons in Millisekunden)
  - **Parameter**: `u16OffTimeMs` = `0`
- **Ereigniseingang**: `REQ` (Startet die Tonausgabe)

Programmablauf und Verbindungen

Der Ablauf der Übung ist sequenziell aufgebaut und wird durch Benutzerinteraktion gestartet:

1. **Start:** Der Benutzer lässt die Softkey-Taste **F1** am Terminal los. Dies wird vom Baustein `SoftKey_UP_F1` erkannt.
2. **Erster Ton:** Das Ereignis `IND` des Softkeys triggert den Eingang `REQ` von `Q_CtrlAudioSignal_0`. Ein Ton mit **440 Hz** wird für 150 ms abgespielt.
3. **Verzögerung:** Sobald der Befehl für den ersten Ton verarbeitet wurde (`CNF`), wird der Verzögerungsbaustein `E_RDELAY` gestartet.
4. **Wartezeit:** Es verstreichen **250 ms** (Parameter `DT`).
5. **Zweiter Ton:** Nach Ablauf der Wartezeit sendet `E_RDELAY` ein Signal über `EO`. Dieses Ereignis aktiviert `Q_CtrlAudioSignal_1`, welcher einen Ton mit **880 Hz** für 150 ms abspielt.

Dies erzeugt eine akustische Rückmeldung in Form einer aufsteigenden Tonfolge (Low-High) mit einer kurzen Pause dazwischen.

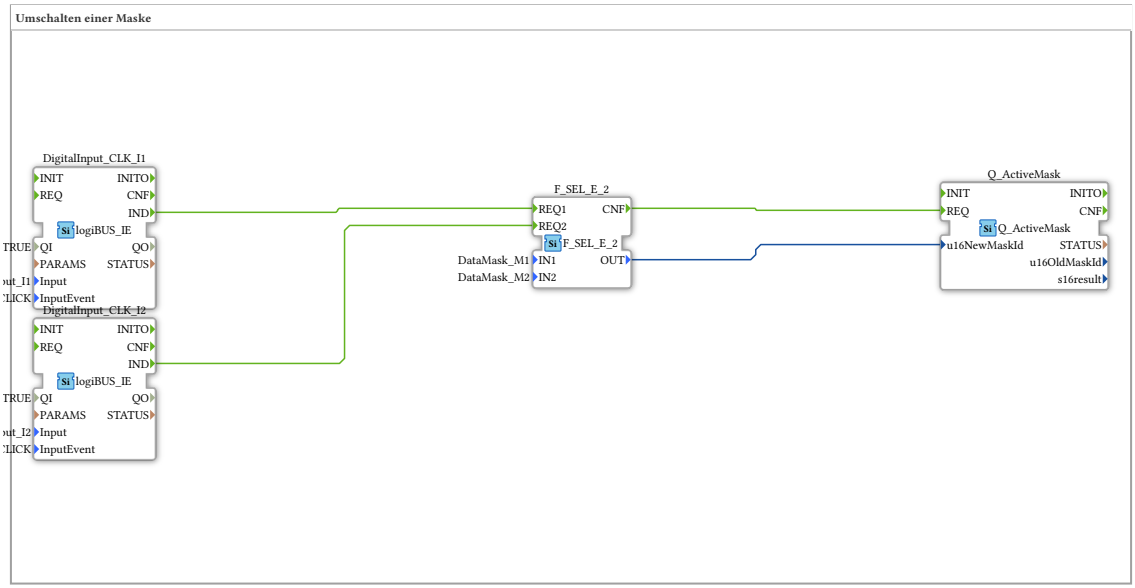
Zusammenfassung

Die Übung `Uebung_018a` vermittelt grundlegende Kenntnisse über die Verkettung von Ereignissen (Event Chaining) in IEC 61499. Sie zeigt praktisch, wie man eine sequentielle Logik aufbaut, bei der eine Aktion (Tonausgabe 1) die nächste Aktion (Verzögerung -> Tonausgabe 2) auslöst, ohne dass der Benutzer erneut eingreifen muss.

2.2.1.185 Uebung_019: Umschalten einer Maske

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_019`. Hier wird gezeigt, wie das Programm die aktive Anzeige (Data Mask) auf dem Terminal umschalten kann.

📺 Video



- Landwirtschaft 1906

🎧 Podcast

- Als Landtechnik-Spezialist durch die Hölle: Wie Lanz-Wery Krieg, Besatzung und Hyperinflation überlebte – Einblicke in Original-Geschäftsberichte 1915-1922
- Land- und Forstwirtschaft 4.0: Das Fundament der Sicherheit – Analyse der DIN EN ISO 25119-1 und der
- RASE: How 19th-Century England Revolutionized Agriculture Through „Practice with Science“
- Rudolf Diesel: Geniales Werk, mysteriöses Ende – Wer verschwand 1913 auf der Fähre?
- Smart Farming Vision 1991 Auernhammers Blaupausen

Ziel der Übung

Verwendung des Bausteins `Q_ActiveMask` zur Navigation auf dem Terminal. Es wird demonstriert, wie physische Taster genutzt werden, um zwischen verschiedenen Bedien-Seiten zu blättern.

Beschreibung und Komponenten

Die Subapplikation `Uebung_019.SUB` nutzt zwei physische Taster, um zwischen zwei Arbeitsmasken zu wählen.

Funktionsbausteine (FBs)

- **`I1` & `I2`**: Physische Eingangs-Taster.
- **`F_SEL_E_2`**: Ein Ereignis-Selektor. Er hat zwei `REQ`-Eingänge und gibt beim jeweiligen Trigger die zugeordnete Konstante am Datenausgang aus.
- **`Q_ActiveMask`**: Der ISOBUS-Ausgangsbaustein. Er sendet den Befehl zum Wechseln der Maske an das Terminal.

Funktionsweise

- Druck auf **Taster 1** ➡️ `F_SEL_E_2` liefert die ID von `DataMask_M1` ➡️ `Q_ActiveMask` schaltet das Terminal auf Seite 1.
- Druck auf **Taster 2** ➡️ `F_SEL_E_2` liefert die ID von `DataMask_M2` ➡️ `Q_ActiveMask` schaltet das Terminal auf Seite 2.

Das System steuert hier also aktiv, was der Bediener sieht.

Anwendungsbeispiel

Kontextabhängige Steuerung: Wenn der Fahrer am physischen Bedienpult den Schalter für „Pflugbetrieb“ umlegt, wechselt das Terminal automatisch von der Straßenansicht zur Feldansicht mit allen relevanten Tiefeneinstellungen. Dies erspart dem Fahrer das manuelle Suchen der richtigen Seite am Touchscreen während der Fahrt.

2.2.1.186 Uebung_019a: Umschalten einer Maske



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_019a`. Hier wird die Maskenumschaltung um eine Sicherheitsfunktion erweitert: Den Alarm.

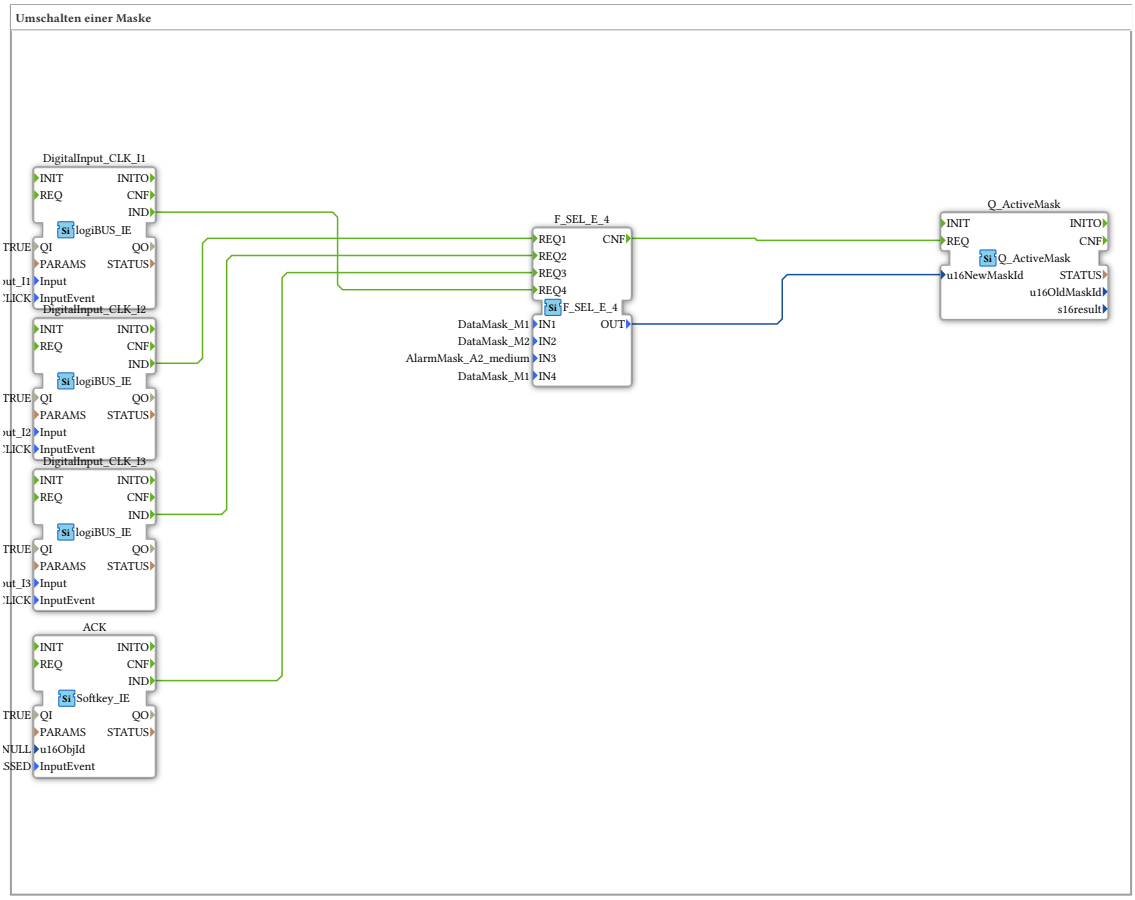
Ziel der Übung

Erlernen des Umgangs mit Alarm-Masken. Im ISOBUS-Standard haben Alarme Vorrang vor normalen Datenmasken und können oft nur durch eine explizite Quittierung (ACK) verlassen werden.

Beschreibung und Komponenten

In `Uebung_019a.SUB` wird ein vierstufiger Selektor (`F_SEL_E_4`) zur Maskenwahl genutzt.

Funktionsbausteine (FBs)



- **`I1` & `I2`**: Normale Maskenwahl (M1, M2).
- **`I3`**: Auslöser für den Alarm.
- **`ACK`**: Ein Softkey am Terminal zum Quittieren des Alarms.
- **`AlarmMask_A2_medium`**: Eine spezielle Alarm-Maske aus dem Pool.

Funktionsweise

1. Durch `I1` und `I2` kann der Nutzer wie gewohnt navigieren.
2. Tritt ein Fehler ein (`I3`), erzwingt die Steuerung die Anzeige der `AlarmMask_A2`. Das Terminal überlagert nun die aktuelle Ansicht mit der Alarmmeldung.
3. Die Navigation über `I1/I2` ist nun wirkungslos oder wird vom Alarm überdeckt (je nach Terminal-Implementierung).
4. Erst wenn der Nutzer am Terminal den Softkey **ACK** drückt, schaltet die Steuerung wieder auf die normale Arbeitsmaske (`M1`) zurück.

Anwendungsbeispiel

Überlastwarnung: Ein Sensor meldet eine drohende Überlastung der Maschine. Die Steuerung unterbricht die normale Anzeige und blendet großformatig die Warnung „Überlast!“ ein. Der Fahrer muss den Fehler bewusst wahrnehmen und am Terminal quittieren, bevor er die normalen Anzeigen wieder nutzen kann.

2.2.1.187 Uebung_019b: Umschalten einer Maske

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_019b`. Hier wird der virtuelle Alarm am Terminal mit einem physischen Alarm-Ausgang synchronisiert.

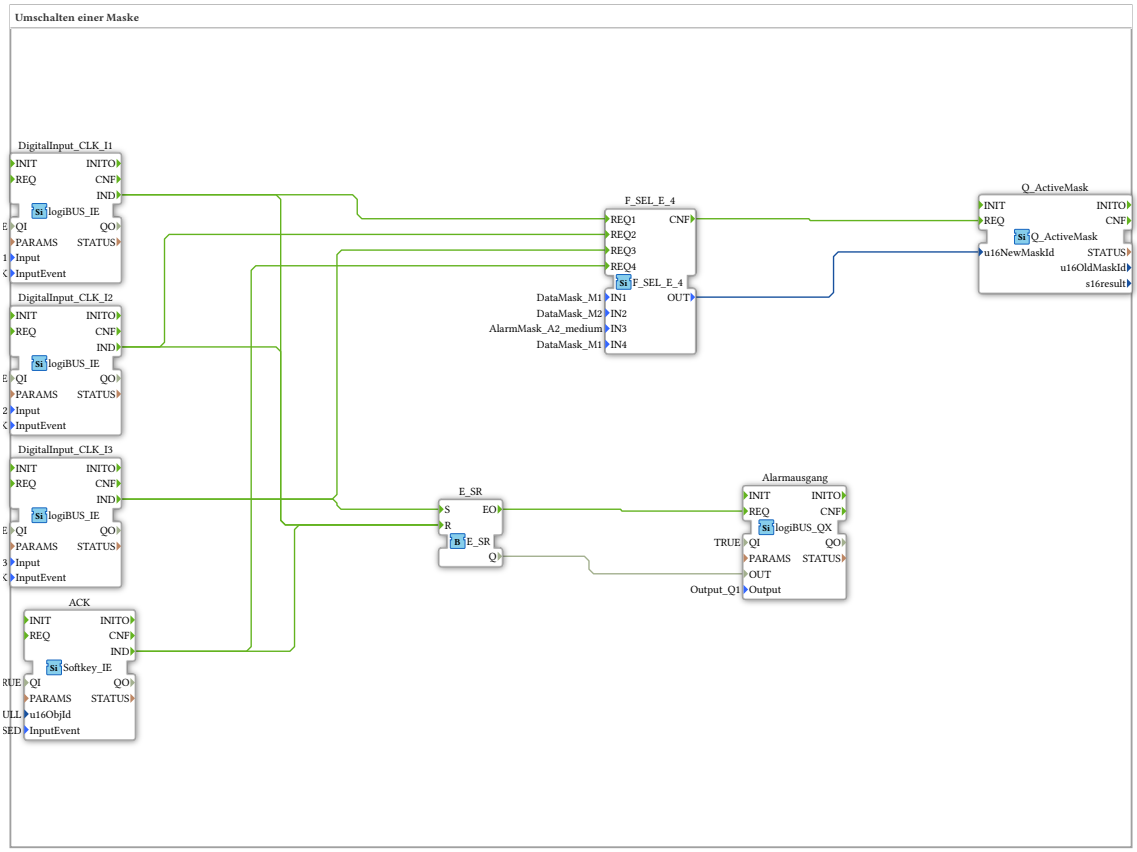
Ziel der Übung

Verknüpfung von UI-Zuständen mit Hardware-Speichern. Es soll sichergestellt werden, dass ein Alarmzustand in der Steuerung erhalten bleibt, bis er am Terminal gelöscht wird.

Beschreibung und Komponenten

In `Uebung_019b.SUB` wird zusätzlich zur Maskenumschaltung ein SR-Flip-Flop für den Alarm-Status verwendet.

Funktionsbausteine (FBs)



- **`E_SR`**: Der Alarm-Speicher.
- **`Alarmausgang`**: Schaltet eine physische Hupe oder Warnlampe (`Q1`).

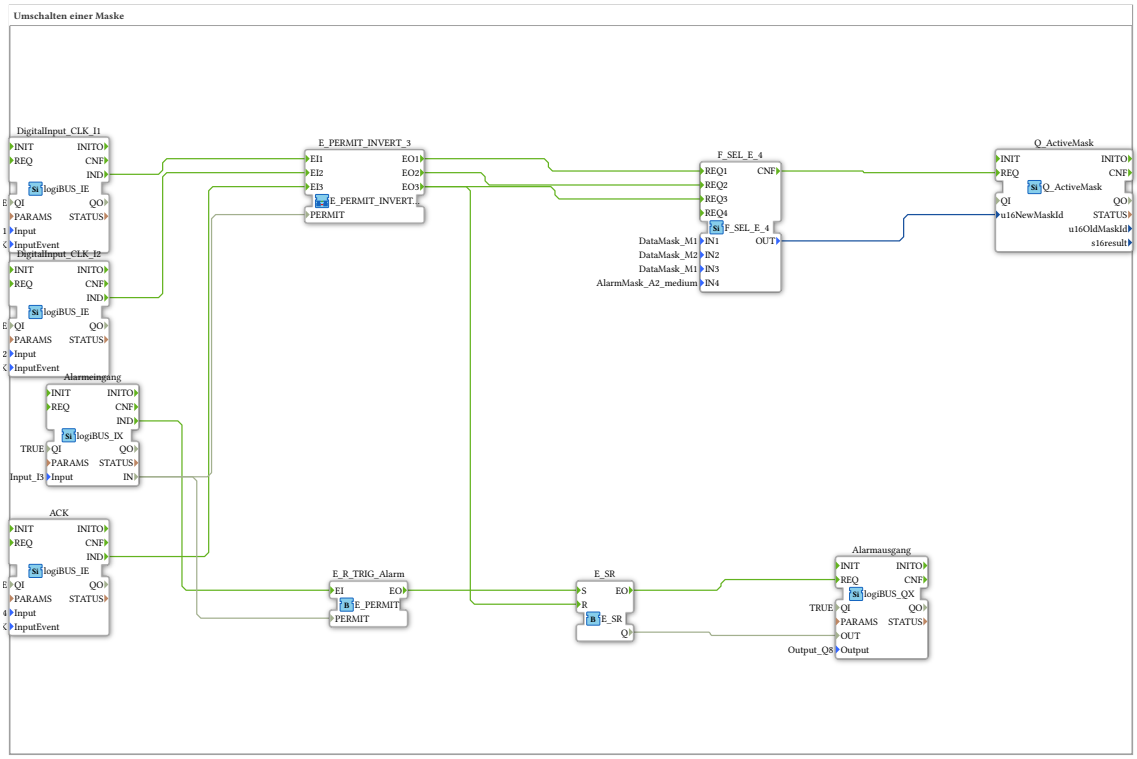
Funktionsweise

- **Alarm auslösen**: Taster `I3` triggert den Alarm. Das Terminal springt auf die Alarmmaske **UND** der Speicher `E_SR` wird gesetzt ➡️ Die physische Hupe geht an.
- **Quittieren**: Der Nutzer drückt **ACK** am Terminal. Die Steuerung wechselt zurück zur normalen Maske **UND** löscht den Speicher `E_SR.R` ➡️ Die Hupe verstummt.
- Interessant: Auch das Wechseln zu einer anderen normalen Maske (`I1`, `I2`) löscht in dieser Implementierung den Alarm-Speicher (Reset-Zweig am `E_SR`).

Anwendungsbeispiel

Zentrale Störmelde-Zentrale: Ein kritischer Fehler (z.B. Öldruckverlust) löst sowohl die Anzeige am Bildschirm als auch eine externe Sirene aus. Der Techniker muss zum Terminal gehen, um zu sehen, was los ist, und durch die Quittierung sowohl das Display aufräumen als auch den Lärm abstellen.

2.2.1.188 Uebung_019c: Umschalten einer Maske



• – ○ • – ○ • – ○ •

Einleitung

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_019c`. Dies ist die komplexeste Navigations-Logik, bei der die Maskenumschaltung aktiv vom Hardware-Zustand blockiert werden kann: Solange der Alarmeingang `I3` aktiv ist, werden die Navigations-Ereignisse nicht an die Maskenauswahl weitergegeben.

Ziel ist die Implementierung einer bedingten Navigationssteuerung. Der Wechsel der Bildschirmseiten soll unterbunden werden, solange ein aktiver Alarm anliegt.

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_CLK_I1**: logiBUS Eingang mit Klick-Erkennung
  - **Typ**: `logiBUS::io::DI::logiBUS_IE`
  - **Parameter**: `QI = TRUE`, `Input = Input_I1`, `InputEvent = BUTTON_SINGLE_CLICK`
  - **Erklärung**: Meldet einen Einzelklick auf `I1` als Event `IND` (Wunsch: Maske 1).
- **DigitalInput_CLK_I2**: logiBUS Eingang mit Klick-Erkennung
  - **Typ**: `logiBUS::io::DI::logiBUS_IE`
  - **Parameter**: `QI = TRUE`, `Input = Input_I2`, `InputEvent = BUTTON_SINGLE_CLICK`
  - **Erklärung**: Meldet einen Einzelklick auf `I2` als Event `IND` (Wunsch: Maske 2).
- **ACK**: physischer Quittier-Taster anstelle eines Softkeys
  - **Typ**: `logiBUS::io::DI::logiBUS_IE`
  - **Parameter**: `QI = TRUE`, `Input = Input_I4`, `InputEvent = BUTTON_SINGLE_CLICK`
  - **Erklärung**: Meldet einen Einzelklick auf `I4` als Event `IND`.
- **Alarmeingang**: physischer Alarm-Sensor
  - **Typ**: `logiBUS::io::DI::logiBUS_IX`
  - **Parameter**: `QI = TRUE`, `Input = Input_I3`
  - **Erklärung**: Liefert den Zustand `IN` von `I3` samt Ereignis `IND`. Solange `IN` TRUE ist, herrscht Alarmzustand.
- **E_PERMIT_INVERT_3**: dreikanaliges, invertiertes Event-Freigabe-Gate (SubApp-Instanz)
  - **Typ**: `MyLib::sys::E_PERMIT_INVERT_3`

  - **Parameter**: keine
  - **Erklärung**: Lässt die Ereignisse `EI1`..`EI3` nur dann als `EO1`..`EO3` durch, wenn `PERMIT` FALSE ist. Intern besteht der Baustein aus `F_NOT_BOOL_INIT` (invertiert `PERMIT`) und `E_PERMIT_3`. Hier wird `PERMIT` vom `Alarmeingang` gespeist: kein Alarm = Ereignisse werden durchgelassen, Alarm = Ereignisse werden blockiert.
- **F_SEL_E_4**: ereignisgesteuerte Auswahl aus vier Werten
  - **Typ**: `eclipse4diac::utils::selection::F_SEL_E_4`
  - **Parameter**: `IN1 = DataMask_M1`, `IN2 = DataMask_M2`, `IN3 = DataMask_M1`, `IN4 = AlarmMask_A2_medium`
  - **Erklärung**: Je nach eingetroffenem Ereignis `REQ1`..`REQ4` wird der zugehörige Wert `IN1`..`IN4` auf `OUT` gelegt und `CNF` gemeldet. Verbunden sind `REQ1`, `REQ2` und `REQ3`; `REQ4` (und damit `AlarmMask_A2_medium`) ist in diesem Netzwerk nicht verdrahtet.
- **Q_ActiveMask**: Wechsel der aktiven Maske
  - **Typ**: `isobus::UT::Q::Q_ActiveMask`
  - **Parameter**: keine
  - **Erklärung**: Setzt die vom `F_SEL_E_4` gelieferte Masken-ID (`u16NewMaskId`) als aktive Maske, ausgelöst durch `REQ`.
- **E_R_TRIG_Alarm**: Freigabe-Gate für den Alarm-Start
  - **Typ**: `iec61499::events::E_PERMIT`
  - **Parameter**: keine
  - **Erklärung**: Leitet das Ereignis `IND` des `Alarmeingang` nur dann als `EO` weiter, wenn `PERMIT` (= `Alarmeingang.IN`) TRUE ist.
- **E_SR**: Alarmspeicher
  - **Typ**: `iec61499::events::E_SR`
  - **Parameter**: keine
  - **Erklärung**: `S` setzt den Speicher (Alarm eingetreten), `R` setzt ihn zurück (quittiert). Ausgang `Q` mit Ereignis `EO`.
- **Alarmausgang**: logiBUS Digitalausgang für die Alarmanzeige
  - **Typ**: `logiBUS::io::DQ::logiBUS_QX`
  - **Parameter**: `QI = TRUE`, `Output = Output_Q8`
  - **Erklärung**: Schaltet den physischen Ausgang `Q8` gemäß `E_SR.Q`.

Programmablauf und Verbindungen

1. `DigitalInput_CLK_I1.IND` ⟶ `E_PERMIT_INVERT_3.EI1`, `DigitalInput_CLK_I2.IND` ⟶ `E_PERMIT_INVERT_3.EI2` und `ACK.IND` ⟶ `E_PERMIT_INVERT_3.EI3`: alle drei Bedien-Ereignisse laufen zuerst durch das invertierte Freigabe-Gate.
2. `Alarmeingang.IN` ⟶ `E_PERMIT_INVERT_3.PERMIT`: das Gate prüft vor jeder Aktion den Alarmeingang.
   - Ist **kein** Alarm vorhanden (`IN = FALSE`), wird das Ereignis durchgelassen: `E_PERMIT_INVERT_3.EO1` ⟶ `F_SEL_E_4.REQ1`, `EO2` ⟶ `REQ2`, `EO3` ⟶ `REQ3`. Die Seite wechselt.
   - Ist ein Alarm aktiv (`IN = TRUE`), wird das Ereignis blockiert. Der Seitenwechsel wird **ignoriert**.
3. `F_SEL_E_4.OUT` ⟶ `Q_ActiveMask.u16NewMaskId` und `F_SEL_E_4.CNF` ⟶ `Q_ActiveMask.REQ`: die gewählte Maske (`DataMask_M1` für `I1`, `DataMask_M2` für `I2`, `DataMask_M1` für `ACK`) wird aktiv.
4. `Alarmeingang.IND` ⟶ `E_R_TRIG_Alarm.EI` und `Alarmeingang.IN` ⟶ `E_R_TRIG_Alarm.PERMIT`: ein Ereignis von `I3` wird nur bei aktivem Alarm (`IN = TRUE`) weitergeleitet.
5. `E_R_TRIG_Alarm.EO` ⟶ `E_SR.S`: der Alarmspeicher wird gesetzt.
6. `E_PERMIT_INVERT_3.EO3` ⟶ `E_SR.R`: der Quittier-Taster `I4` setzt den Speicher zurück, aber nur, wenn der Alarmeingang `I3` wieder FALSE ist (sonst blockiert das Gate das Ereignis).
7. `E_SR.EO` ⟶ `Alarmausgang.REQ` und `E_SR.Q` ⟶ `Alarmausgang.OUT`: der Ausgang `Q8` folgt dem Alarmspeicher.

Verhalten: Solange `I3` aktiv ist, bleiben `I1`, `I2` und `I4` wirkungslos. Erst wenn der Alarm-Sensor (`I3`) wieder FALSE ist **UND** der Nutzer den Quittier-Taster (`I4`) drückt, wird der Speicher zurückgesetzt und gleichzeitig auf `DataMask_M1` gewechselt; die Navigation ist dann wieder frei.

Zusammenfassung

Die Übung zeigt, wie ein invertiertes Event-Freigabe-Gate (`E_PERMIT_INVERT_3`) die Maskenumschaltung vom Hardware-Zustand abhängig macht: ein aktiver Alarm sperrt die Navigation, und erst das Quittieren bei zurückgenommenem Alarm gibt sie wieder frei.

Anwendungsbeispiel - zwingende Störungsbehebung: Bei einem kritischen Hardware-Fehler (z.B. Not-Aus betätigt) darf der Bediener das Terminal nicht weiter zur normalen Maschinensteuerung nutzen.

Er wird auf der Alarmseite „festgehalten“, bis der Not-Aus entriegelt und die Störung quittiert wurde. Dies erzwingt die Aufmerksamkeit für das vorrangige Sicherheitsproblem.

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

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

2.2.1.189 Uebung_020a: DigitalInput_I1 auf DigitalOutput_Q1

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_020a`. Hier wird die manuelle Erzeugung von Setz- und Rücksetz-Ereignissen aus einem Standard-Datensignal demonstriert.

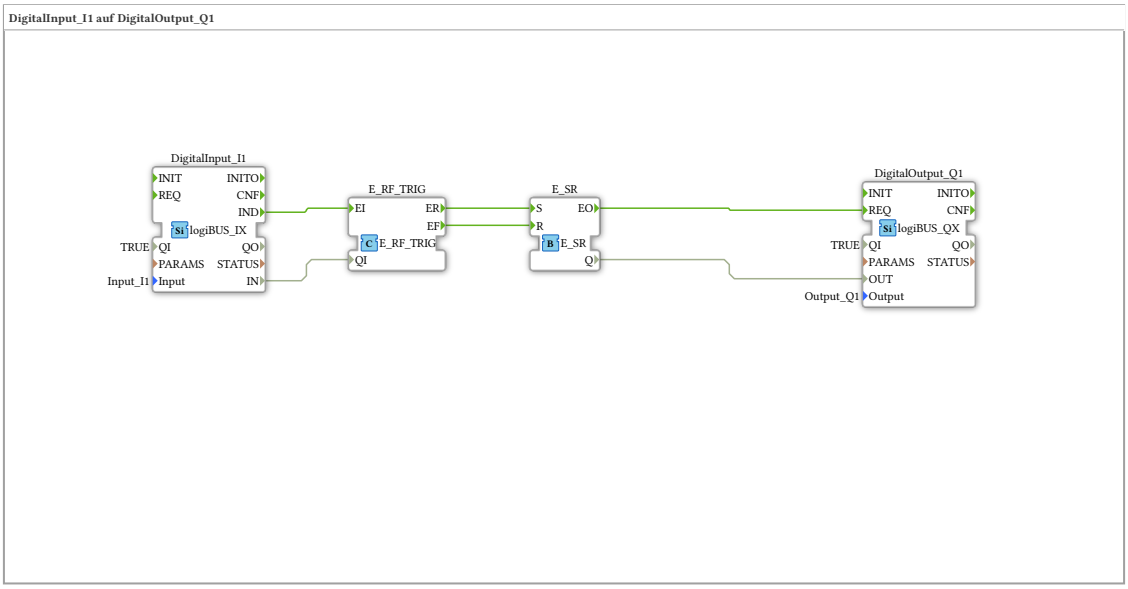
Ziel der Übung

Verständnis der Flankenerkennung. Es wird gezeigt, wie man mit einer Ereignis-Weiche (`E_SWITCH`) und einem Speicher (`E_RS`) ein Verhalten realisiert, bei dem der Ausgang beim Drücken eines Tasters eingeschaltet und beim Loslassen ausgeschaltet wird (entspricht funktional einer direkten Leitung, aber mit expliziter Logik-Trennung).

Beschreibung und Komponenten

Die Subapplikation `Uebung_020a.SUB` nutzt einen `logiBUS_IX` Eingang, um einen `E_RS` Speicher zu steuern.

Funktionsbausteine (FBs)



- **`DigitalInput_I1`**: Standard-Eingang. Liefert ein Event bei jeder Änderung.
- **`E_SWITCH`**: Leitet das Event je nach Pegel an `S` oder `R` weiter.
- **`E_RS`**: Der Ereignis-Speicher.

Funktionsweise

1. **Drücken (TRUE)**: Das `IND`-Event geht zur Weiche. Da `G=TRUE`, feuert `EO1` ➡️ `E_RS.S` (Setzen).
2. **Loslassen (FALSE)**: Das `IND`-Event geht zur Weiche. Da `G=FALSE`, feuert `EO0` ➡️ `E_RS.R` (Rücksetzen).

Obwohl das Ergebnis eine 1:1 Abbildung des Eingangs ist, zeigt diese Übung den inneren Mechanismus von speichernden Systemen.

2.2.1.190 Uebung_020b: DigitalInput_I1 auf DigitalOutput_Q1; TON Einschaltverzögert; aufgelöst

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_020b`. Hier wird eine Einschaltverzögerung (TON) manuell aus Grundbausteinen aufgebaut.

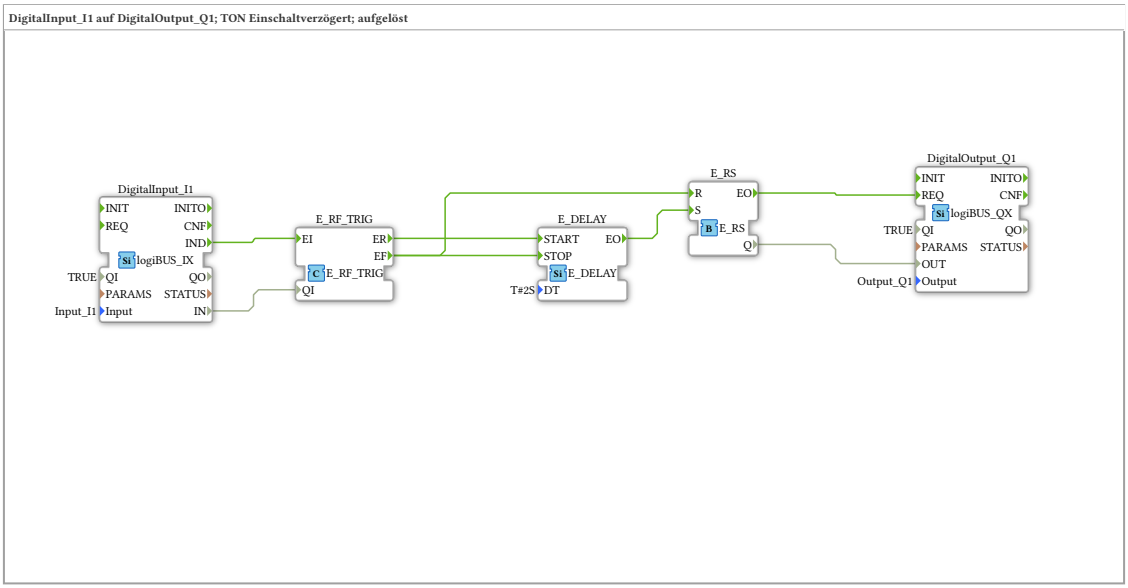
Ziel der Übung

Verstehen der Zeitsteuerung durch Ereignisverzögerung (`E_DELAY`). Es wird gezeigt, wie ein Timer-Verhalten („Licht geht erst nach 2 Sekunden an“) durch das gezielte Verzögern und Abbrechen von Ereignissen realisiert wird.

Beschreibung und Komponenten

In `Uebung_020b.SUB` wird ein Verzögerungs-Baustein zwischen die Eingangs-Weiche und den Speicher geschaltet.

Funktionsbausteine (FBs)



- **`E_DELAY`**: Wartet die Zeit `DT` (2 Sekunden) ab.
- **`E_SWITCH`**: Steuert den Start und Stopp des Timers.

Funktionsweise

1. **Start**: Nutzer drückt `I1`. Die Weiche schaltet auf `EO1` ➡️ `E_DELAY.START`.
2. **Warten**: Wenn der Nutzer den Taster für volle 2 Sekunden gedrückt hält, feuert `E_DELAY.EO` ➡️ `E_RS.S`. Die Lampe geht an.
3. **Abbruch**: Lässt der Nutzer den Taster vor Ablauf der 2 Sekunden los, schaltet die Weiche auf `EO0`. Dieses Event triggert `E_DELAY.STOP` (Timer wird gelöscht) **und** `E_RS.R` (Ausgang wird sicher auf FALSE gesetzt).

Ergebnis: Einschaltverzögerung mit sofortigem Abbruch bei Signalverlust.

2.2.1.191 Uebung_020c: DigitalInput_I1 auf DigitalOutput_Q1; E_TON; Einschaltverzögert



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_020c`.

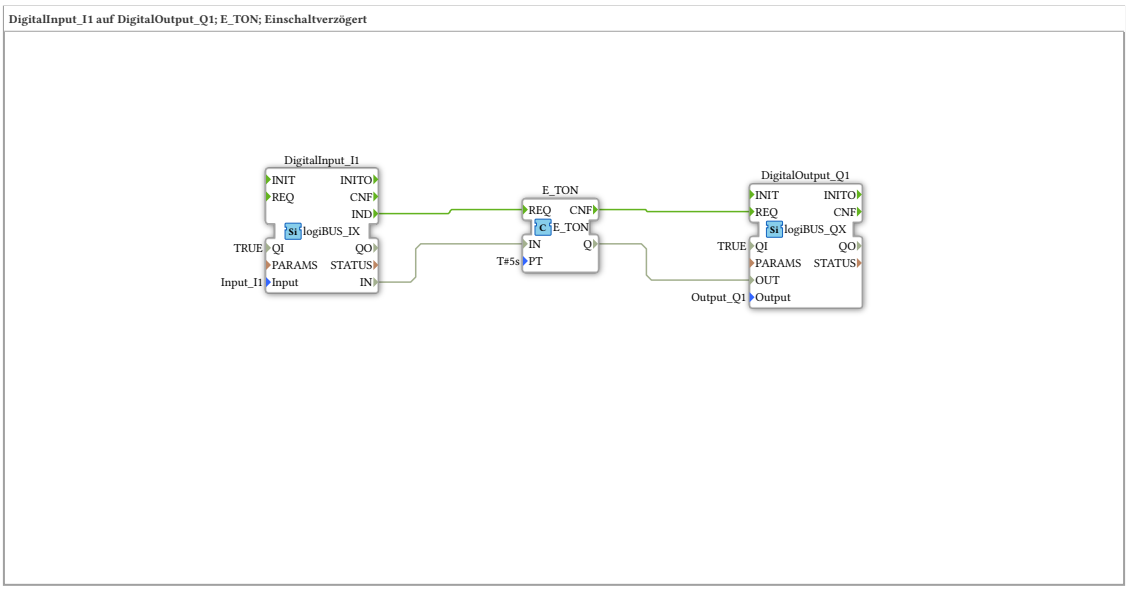
Ziel der Übung

Nutzung des standardisierten Timer-Bausteins `E_TON`.

Beschreibung und Komponenten

Die Subapplikation `Uebung_020c.SUB` nutzt den `E_TON` Baustein aus der Event-Timer-Bibliothek.

Funktionsbausteine (FBs)



- **`E_TON`**: Timer ON-Delay (Ereignisbasiert).
- **Parameter `PT`**: Preset Time (hier 5 Sekunden).

Funktionsweise

Der Baustein vereinfacht den Aufbau aus Übung 020b erheblich:

- Eingang `I1` wird TRUE ➡️ Timer startet.
- Nach 5s wird der Ausgang `Q` TRUE.
- Eingang `I1` wird FALSE ➡️ Timer bricht ab, Ausgang wird sofort FALSE.

Dies ist der Standardweg, um Verzögerungen in 4diac zu realisieren.

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

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

2.2.1.192 Uebung_020c2: DigitalInput_I1 auf DigitalOutput_Q1; E_TON Einschaltverzögert; Zeit eingeben und speichern



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_020c2`. Hier wird die Einschaltverzögerung mit einer Benutzereingabe am Terminal und Datenspeicherung kombiniert.

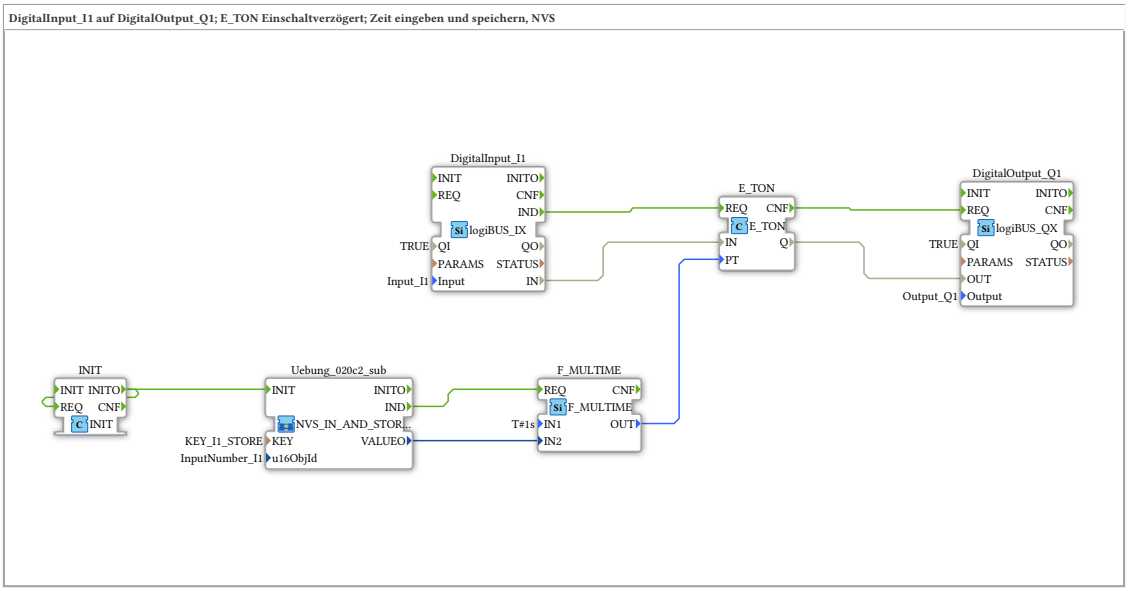
Ziel der Übung

Dynamische Anpassung von Timer-Zeiten zur Laufzeit.

Beschreibung und Komponenten

In `Uebung_020c2.SUB` wird die Verzögerungszeit (`PT`) nicht fest im Programm hinterlegt, sondern vom ISOBUS-Terminal eingelesen.

Funktionsbausteine (FBs)



- **`Uebung_020c2_sub`**: Eine Speicher-SubApp (wie in Übung 012a), die den vom Nutzer eingegebenen Zahlenwert verwaltet.
- **`F_MULTIME`**: Multipliziert einen Zeitwert. Hier wird der Zahlenwert (z.B. „5“) mit der Einheit `T#1s` multipliziert, um den Datentyp `TIME` für den Timer zu erzeugen (z.B. 5 Sekunden).
- **`E_TON`**: Der eigentliche Verzögerungs-Baustein.

Funktionsweise

1. Der Nutzer gibt am Terminal eine „5“ ein.
2. Der Wert wird im NVS gespeichert und an die Logik übergeben.
3. `F_MULTIME` macht daraus 5 Sekunden.
4. Wird nun der physische Taster `I1` gedrückt, verzögert `E_TON` das Signal um exakt diese 5 Sekunden.

Ändert der Nutzer den Wert am Terminal auf „10“, reagiert der Timer ab sofort mit 10 Sekunden Verzögerung.

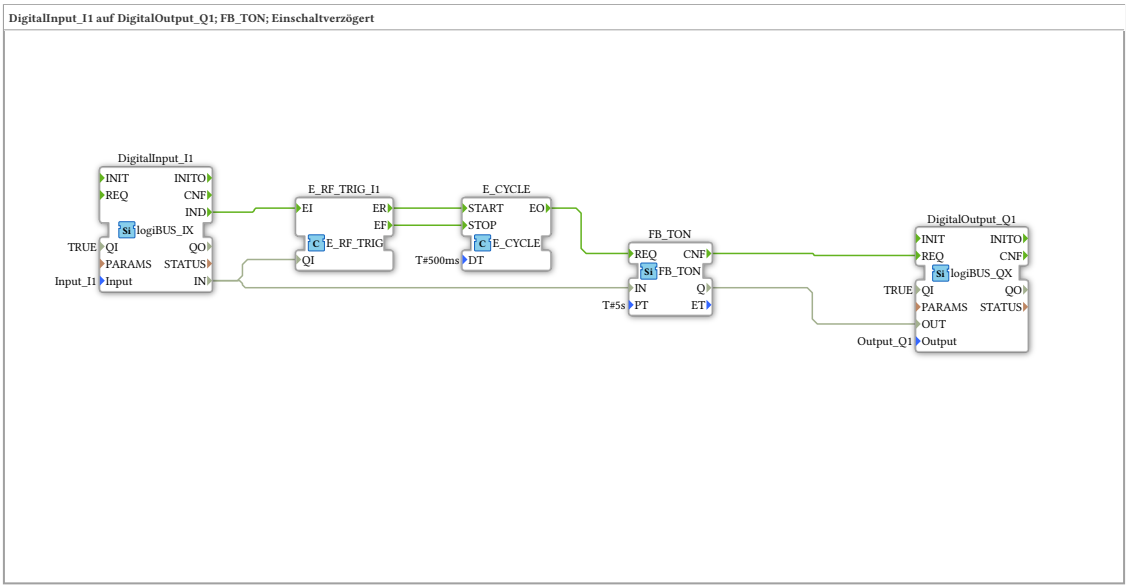
2.2.1.193 Uebung_020c2_sub

Hinweis

Diese Übung existiert in den aktuellen Trainingsquellen nicht mehr (entfernt, zusammengelegt oder umbenannt). Diese Seite bleibt nur als Platzhalter für bestehende Verweise erhalten. Die aktuellen Übungen stehen in der Übungsübersicht.

2.2.1.194 Uebung_020c3: DigitalInput_I1 auf DigitalOutput_Q1; FB_TON; Einschaltverzögert

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_020c3`. Hier wird der klassische IEC 61131-3 Timer-Baustein `FB_TON` verwendet, der eine regelmäßige Triggerung (Takt) benötigt.



Wichtiger Hinweis: Dieser Baustein funktioniert nur korrekt, wenn er zyklisch aufgerufen wird.

Ziel der Übung

Das Ziel ist es, eine Einschaltverzögerung mit einem klassischen SPS-Verhalten (inkl. ET-Ausgang) in einer ereignisbasierten Umgebung zu realisieren. Im Gegensatz zum ereignisbasierten `E_TON` benötigt der `FB_TON` einen regelmäßigen Trigger (Abtastung), um seine interne Zeitrechnung und den Ausgang `ET` zu aktualisieren.

Beschreibung und Komponenten

In `Uebung_020c3.SUB` wird ein Taktgeber verwendet, um den klassischen Timer anzutreiben.

Funktionsbausteine (FBs)

- **`FB_TON`**: Der klassische TON-Baustein.
- **`E_CYCLE`**: Ein Zeitgeber, der alle 500ms ein Ereignis an den `REQ`-Eingang des Timers sendet.

Funktionsweise

Damit der `FB_TON` korrekt funktioniert, muss er „befragt“ werden.

1. Der Nutzer drückt Taster `I1`. Das Signal liegt am Dateneingang `IN` des Timers an.
2. Gleichzeitig startet der Taster über eine Weiche den `E_CYCLE`.
3. Alle 500ms fordert der Cycle den Timer zur Berechnung auf (`REQ`).
4. Erst bei diesen Abfragen prüft der Timer, wie viel Zeit vergangen ist.
5. Sobald die 5 Sekunden erreicht sind, wechselt der Ausgang `Q` auf TRUE.

Diese Methode ist notwendig, wenn man Bausteine aus der 61131-Welt in die 61499-Event-Welt integriert.

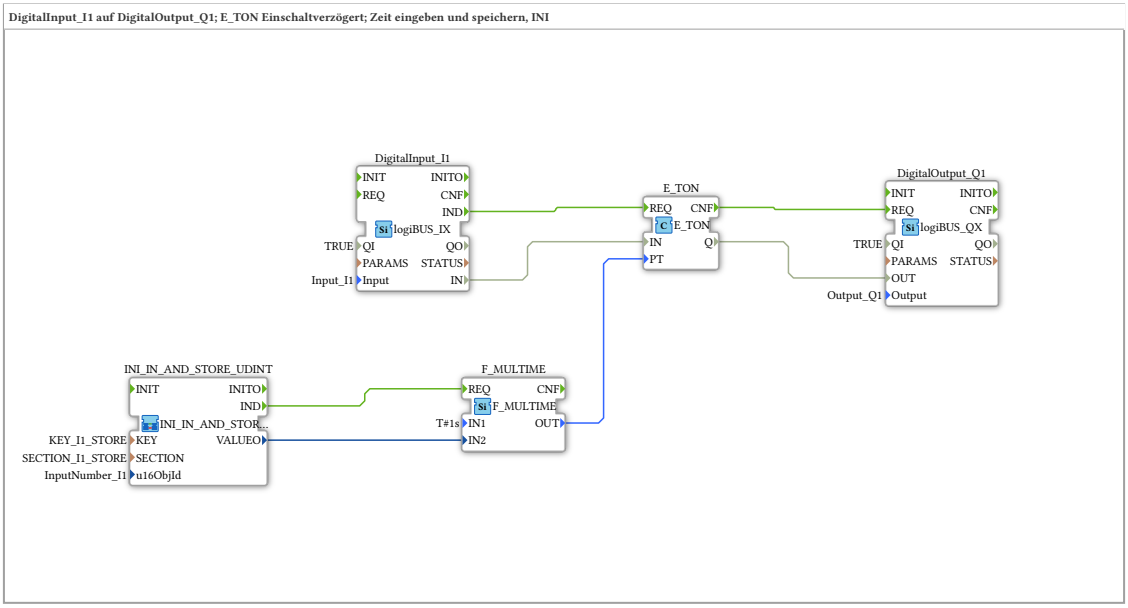
2.2.1.195 Uebung_020c4: Einschaltverzögerung mit einstellbarer, INI-gespeicherter Zeit

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_020c4`. Direkte (nicht-AX-Adapter-) Variante von `Uebung_020c4_AX` - siehe dort für die konzeptionell identische Funktionsweise.

Ziel der Übung

`DigitalInput_I1` schaltet `DigitalOutput_Q1` erst nach einer einstellbaren Verzögerung (`E_TON`). Die Verzögerungszeit ist über das VT-Terminal einstellbar und wird in einer INI-Datei (Abschnitt `SECTION_I1_STORE`, Schlüssel `KEY_I1_STORE`) dauerhaft gespeichert.

Beschreibung und Komponenten



- **`DigitalInput_I1`**: `logiBUS::io::DI::logiBUS_IX`.
- **`E_TON`**: `iec61499::events::timers::E_TON`, Einschaltverzögerungs-Timer.
- **`F_MULTIME`**: `iec61131::arithmetic::F_MULTIME`, berechnet die tatsächliche `PT` (Preset Time) aus dem gespeicherten Wert.
- **`INI_IN_AND_STORE_UDINT`**: liest den vom Bediener eingegebenen Wert und speichert ihn in der INI-Datei.
- **`DigitalOutput_Q1`**: `logiBUS::io::DQ::logiBUS_QX`.

Funktionsweise

1. Der Bediener gibt am ISOBUS-Terminal einen Zahlenwert ein, der über `INI_IN_AND_STORE_UDINT` sofort in die INI-Datei geschrieben wird.
2. `F_MULTIME` berechnet daraus die tatsächliche Verzögerungszeit (`PT`) für `E_TON`.
3. Sobald `DigitalInput_I1` aktiv wird (`IND`), startet `E_TON` die Verzögerung.
4. Nach Ablauf der gespeicherten Zeit schaltet `DigitalOutput_Q1` ein (`REQ`).
5. Die gespeicherte Zeit übersteht einen Neustart des Controllers.

2.2.1.196 Uebung_020d: DigitalInput_I1 auf DigitalOutput_Q1; TOF Ausschaltverzögert; aufgelöst

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_020d`. Hier wird die Funktion einer Ausschaltverzögerung (TOF) manuell aus Grundbausteinen aufgebaut.

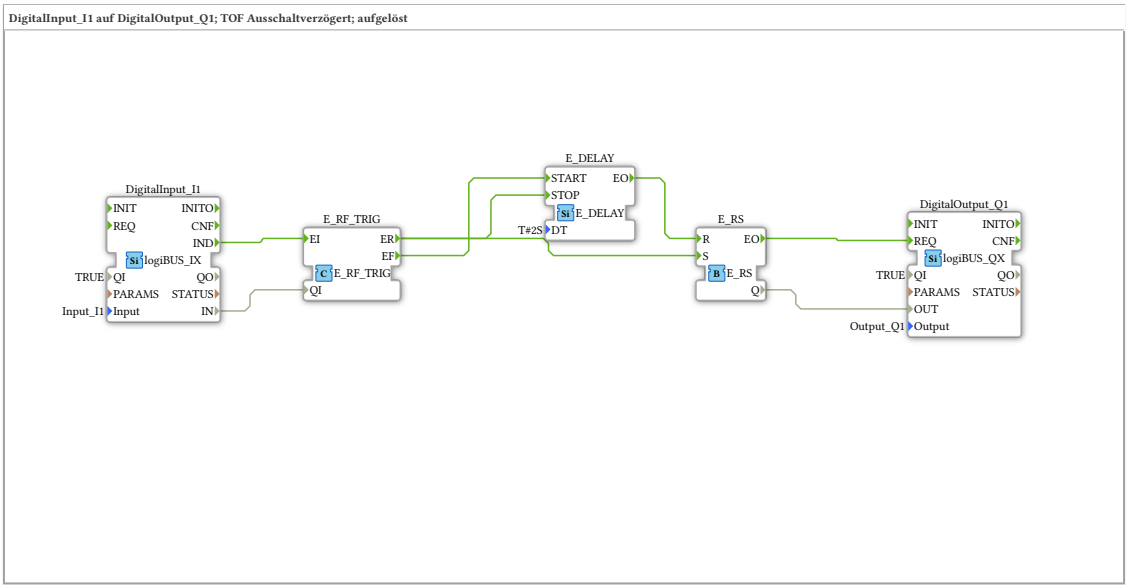
Ziel der Übung

Realisierung eines Nachlauf-Verhaltens. Der Ausgang soll beim Drücken des Tasters sofort angehen, aber nach dem Loslassen noch eine definierte Zeit (2 Sekunden) aktiv bleiben.

Beschreibung und Komponenten

In `Uebung_020d.SUB` wird die TOF-Logik durch geschickte Verknüpfung von `E_DELAY` und `E_RS` implementiert.

Funktionsweise



1. **Einschalten**: Nutzer drückt `I1`. Die Weiche leitet das Event an `EO1`. Dies bewirkt zwei Dinge:
   - Der Speicher `E_RS` wird sofort gesetzt (Lampe geht an).
   - Ein eventuell noch laufender Verzögerungs-Timer wird gestoppt (`E_DELAY.STOP`).
2. **Halten**: Solange gedrückt wird, bleibt der Zustand stabil.
3. **Ausschalten**: Nutzer lässt `I1` los. Die Weiche schaltet auf `EO0`. Dies triggert den Verzögerungs-Timer (`E_DELAY.START`).
4. **Nachlauf**: Erst wenn die 2 Sekunden abgelaufen sind, feuert der Timer `E_DELAY.EO` ➡️ `E_RS.R`. Der Speicher wird zurückgesetzt, die Lampe geht aus.

Anwendungsbeispiel

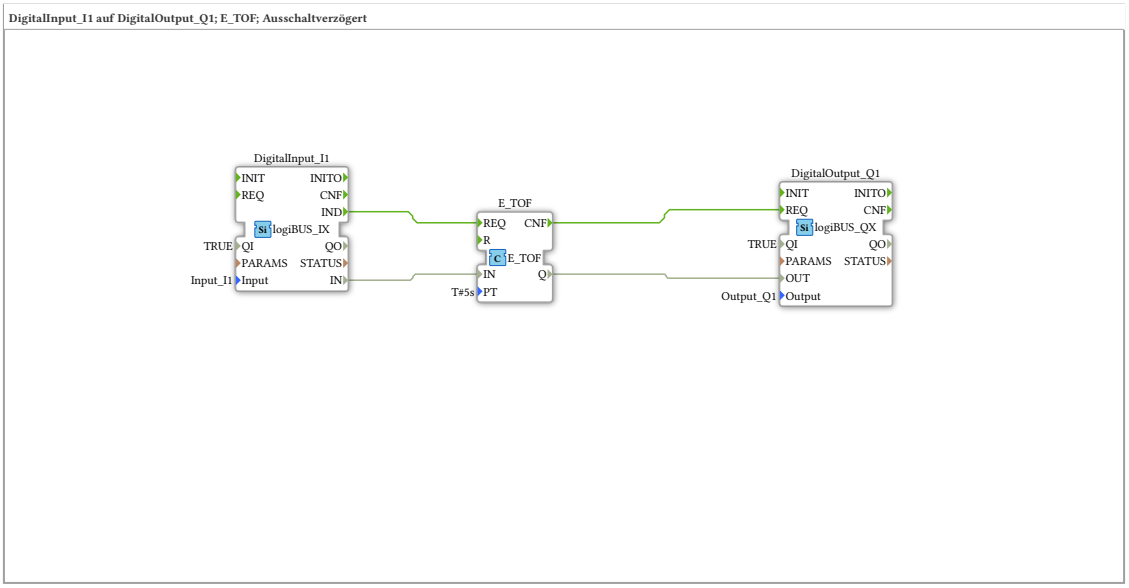
Innenraum-Beleuchtung im Auto: Sobald die Tür geöffnet wird (`I1`), geht das Licht an. Wird die Tür geschlossen, bleibt das Licht noch einige Sekunden hell, damit man sich anschnallen kann, und geht dann erst aus.

2.2.1.197 Uebung_020e: DigitalInput_I1 auf DigitalOutput_Q1; E_TOF; Ausschaltverzögert

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_020e`.

Podcast



- Infineon BTM9020EP Vollbrücke verstehen
- integrierten Vollbrücken-ICs MOTIX™ BTM9020EP

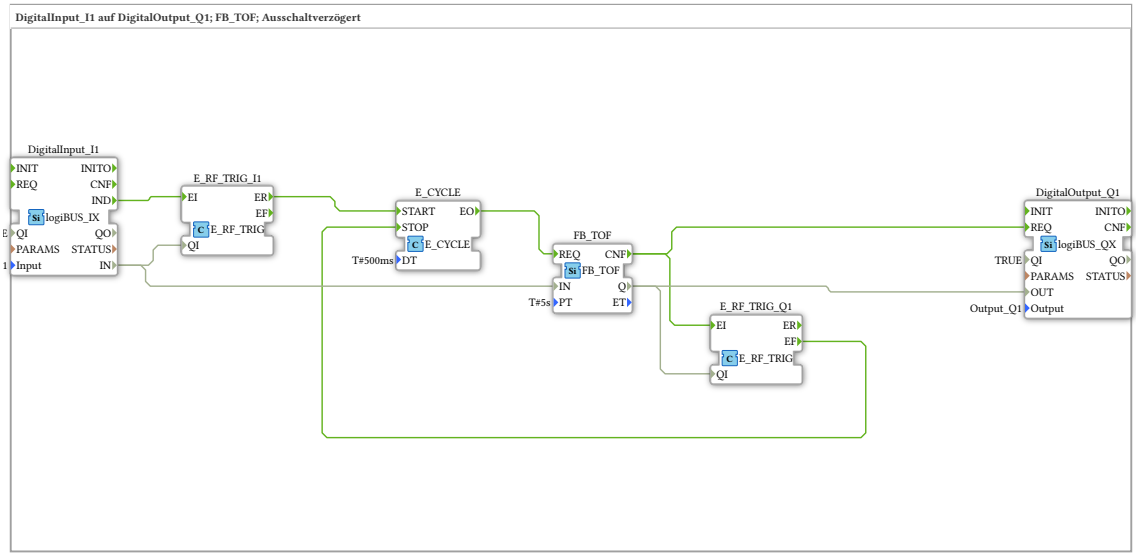
Übersicht

Verwendung des standardisierten ereignisbasierten Timers `E_TOF`. Die Logik entspricht der Übung 020d, ist aber in einem einzigen Baustein gekapselt. Ein Signal am Eingang `IN` wird sofort zum Ausgang `Q` durchgereicht. Fällt `IN` weg, bleibt `Q` noch für die Zeit `PT` (hier 5 Sekunden) auf `TRUE`.

2.2.1.198 Uebung_020e2: DigitalInput_I1 auf DigitalOutput_Q1; FB_TOF; Ausschaltverzögert



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_020e2`. Hier wird der klassische IEC 61131-3 Timer-Baustein `FB_TOF` verwendet, der eine regelmäßige Triggerung (Takt) benötigt.



Wichtiger Hinweis: Dieser Baustein funktioniert nur korrekt, wenn er zyklisch aufgerufen wird.

Übersicht

Demonstration des klassischen `FB_TOF` Bausteins. Da dieser Baustein zyklische Abfragen benötigt, wird er wie in Übung 020c3 über einen `E_CYCLE` (hier 500ms) angetrieben. Zusätzlich sorgt ein zweiter `E_SWITCH` am Ausgang dafür, dass der Taktgeber `E_CYCLE` gestoppt wird, sobald die Nachlaufzeit beendet ist.

2.2.1.199 Uebung_020f: DigitalInput_I1 auf DigitalOutput_Q1; E_TP; Impulsformend

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_020f`.

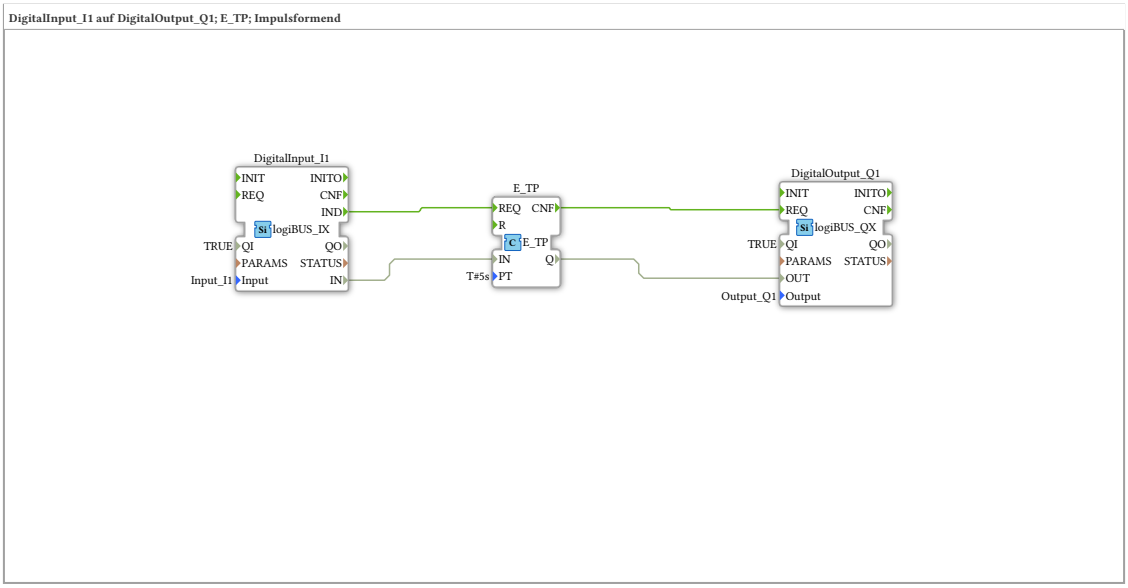
Ziel der Übung

Nutzung des impulsformenden Timers `E_TP` (Timer Pulse).

Funktionsweise

Sobald der Eingang `IN` auf `TRUE` wechselt, schaltet der Ausgang `Q` für exakt die Zeit `PT` (hier `5` Sekunden) ein. Das Besondere: Der Ausgang bleibt für die volle Zeit aktiv, auch wenn der Eingang `IN` zwischendurch wieder abfällt oder mehrfach gedrückt wird (nicht re-triggerbar).

Anwendungsbeispiel



Türöffner: Ein kurzer Druck auf den Taster lässt den elektrischen Türöffner für `5` Sekunden summen, damit der Gast eintreten kann. Die Zeit läuft unabhängig davon ab, wie lange der Bewohner den Taster tatsächlich gedrückt hält.

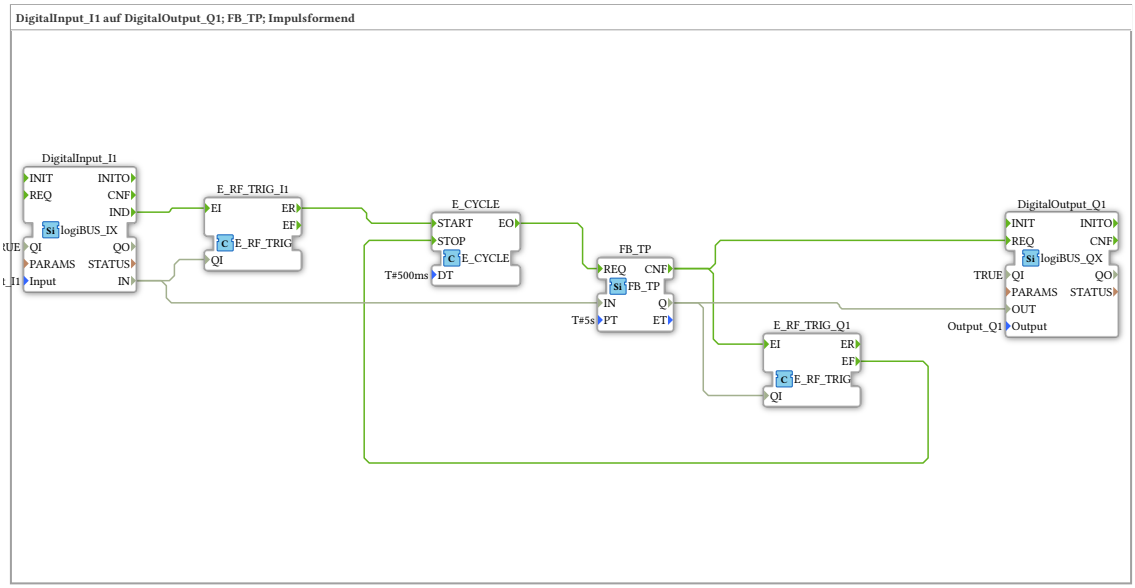
2.2.1.200 Uebung_020f2: DigitalInput_I1 auf DigitalOutput_Q1; FB_TP; Impulsformend

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_020f2`. Hier wird der klassische IEC 61131-3 Timer-Baustein `FB_TP` verwendet.

Übersicht

Diese Übung implementiert einen Impulsgeber unter Verwendung des klassischen `FB_TP` Bausteins. Da dieser Baustein für eine zyklische SPS-Umgebung entworfen wurde, muss in der ereignisbasierten IEC 61499 ein `E_CYCLE` zur regelmäßigen Triggerung genutzt werden.

Funktionsweise



1. **Trigger**: Die steigende Flanke von `Input_I1` startet über einen `E_SWITCH` den Taktgeber `E_CYCLE`.
2. **Berechnung**: Der `E_CYCLE` triggert alle 500ms den `REQ`-Eingang des `FB_TP`. Nur so kann der Timer intern die Zeit hochzählen und den Ausgang `ET` (Elapsed Time) aktualisieren.
3. **Abschluss**: Sobald der Impuls beendet ist (`Q` geht auf `FALSE`), stoppt der `E_CYCLE` automatisch, um unnötige CPU-Last zu vermeiden.

⚖️ Vergleich zur AX-Variante

Im Gegensatz zur `AX_FB_TP` Variante (Übung 020f2_AX) werden hier klassische boolesche Ein- und Ausgänge verwendet, anstatt auf die flexibleren AX-Adapter zu setzen. Die zugrunde liegende Problematik des zyklischen Aufrufs bleibt jedoch identisch.

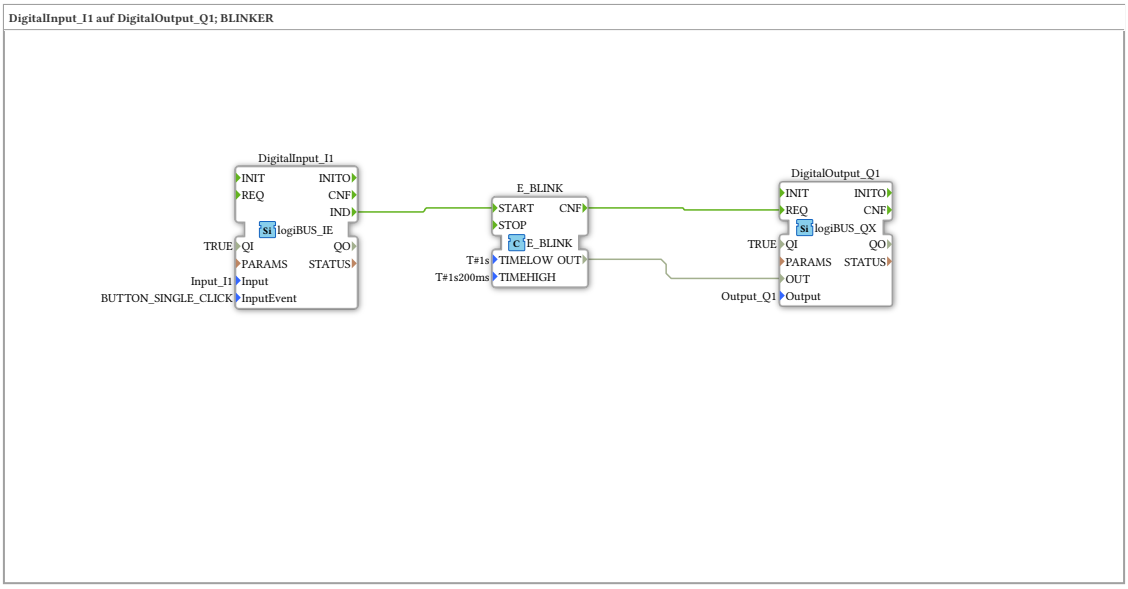
2.2.1.201 Uebung_020f3: DigitalInput_I1 auf DigitalOutput_Q1; BLINKER

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_020f3`.

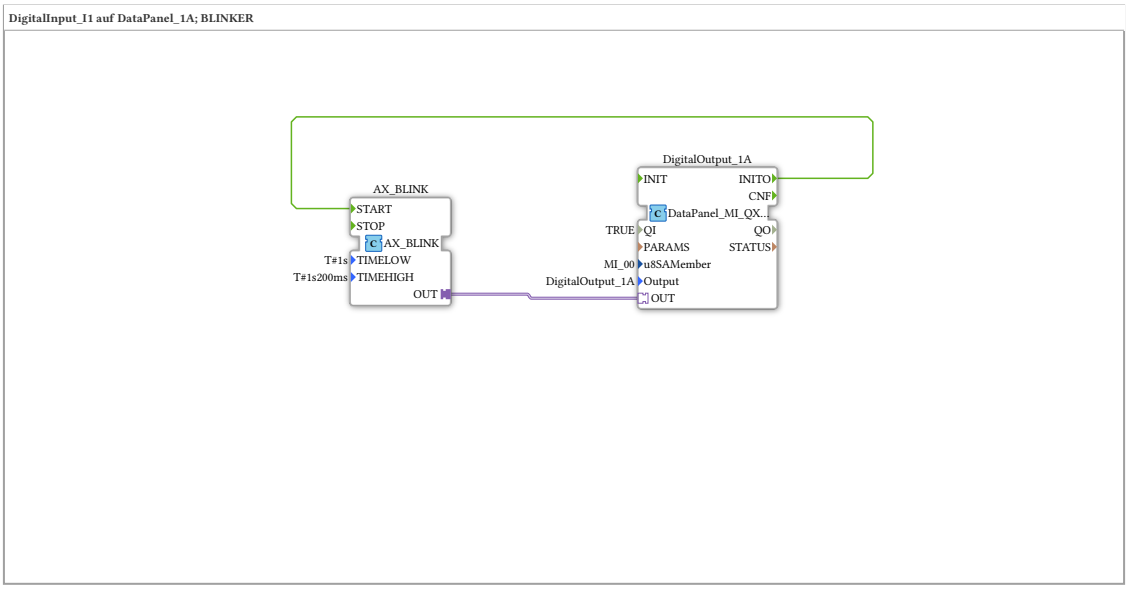
Übersicht

Verwendung des spezialisierten Blinker-Bausteins `E_BLINK`. Dieser Baustein fasst die gesamte Logik der Übung 007a3 zusammen. Über getrennte Parameter für `TIMELOW` und `TIMEHIGH` können asymmetrische Blinkmuster (z.B. kurzes Blitzen) einfach realisiert werden. Ein Ereignis am Eingang `START` setzt den Blinker in Gang.



2.2.1.202 Uebung_020f4: DigitalInput_I1 auf DataPanel_1A; BLINKER

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_020f4 (DigitalInput_I1 auf DataPanel_1A; BLINKER).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **DigitalInput_I1 auf DataPanel_1A; BLINKER**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_020f4.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **AX_BLINK**: Instanz des Typs adapter::events::unidirectional::signals::AX_BLINK.
  - Parameter TIMELOW = T#1s
  - Parameter TIMEHIGH = T#1s200ms
- **DigitalOutput_1A**: Instanz des Typs DataPanel::io::MI::DQ::DataPanel_MI_QXA.
  - Parameter QI = TRUE

- Parameter u8SAMember = MI_00
- Parameter Output = DigitalOutput_1A

Verbindungen und Schnittstellen

Adapterverbindungen:

- AX_BLINK.OUT -> DigitalOutput_1A.OUT

Ereignisverbindungen:

- DigitalOutput_1A.INITO -> AX_BLINK.START

Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_020f4 demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.2.1.203 Uebung_020g: DigitalInput_I1 auf DigitalOutput_Q1; E_TONOF; Ein-/Ausschaltverzögert

NotebookLM

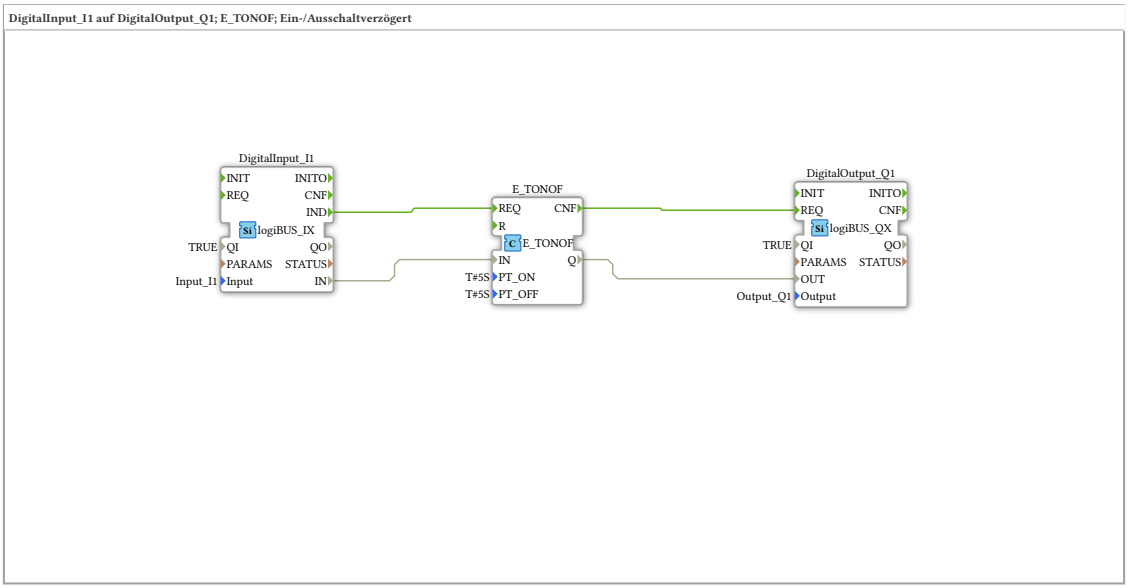
Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_020g`.

Ziel der Übung

Verwendung des Bausteins `E_TONOF`, der sowohl eine Einschalt- als auch eine Ausschaltverzögerung in einem Gehäuse bietet.

Funktionsweise

Der Baustein reagiert auf den Pegel am Eingang `IN`:



- Wechsel zu `TRUE`: Der Ausgang wird erst nach Ablauf von `PT_ON` (5s) aktiv.
- Wechsel zu `FALSE`: Der Ausgang bleibt noch für die Zeit `PT_OFF` (5s) aktiv.

Dies filtert kurze Impulse (Störungen) am Eingang komplett heraus und sorgt gleichzeitig für einen definierten Nachlauf.

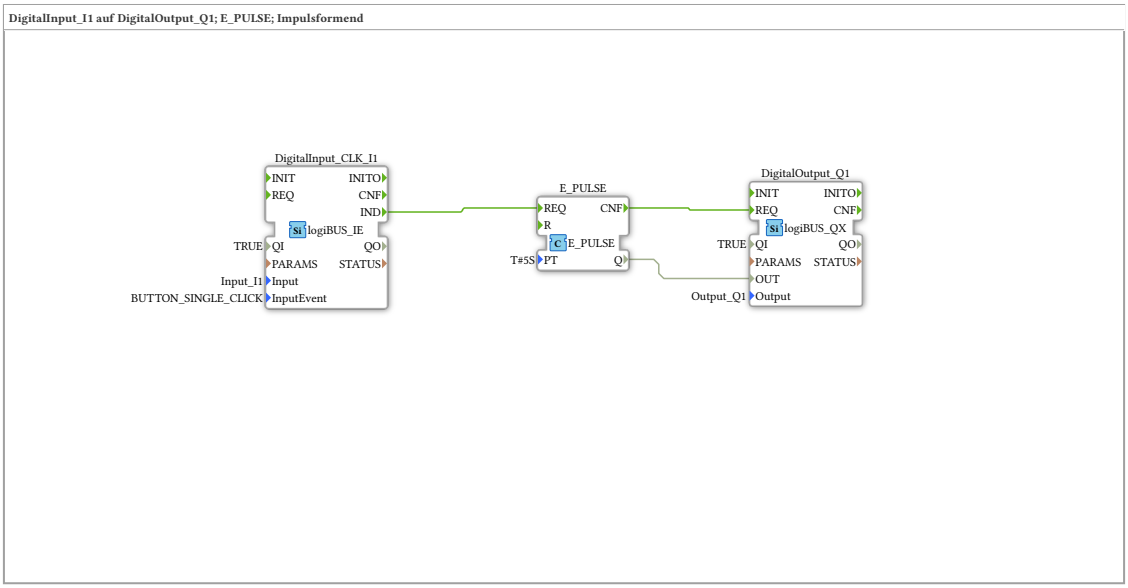
2.2.1.204 Uebung_020h: DigitalInput_I1 auf DigitalOutput_Q1; E_PULSE; Impulsformend



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_020h`.

Übersicht

Diese Übung zeigt die Ansteuerung des Bausteins `E_PULSE` durch einen Ereignis-Eingang (`logiBUS_IE`). Jeder erkannte Einzelklick am Taster löst am Ausgang einen exakt 5 Sekunden langen Impuls aus. Da `E_PULSE` ein reiner Event-Baustein ist, benötigt er kein dauerhaftes Datensignal am Eingang, sondern nur den Start-Trigger.



2.2.1.205 Uebung_020i: DigitalInput_I1 auf DigitalOutput_Q1; E_PULSE; Impulsformend; Teach-In



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_020i`. Dies ist eine sehr praxisnahe Übung, bei der eine Zeitdauer nicht durch Zahlenwerte, sondern durch „Vormachen“ (Teach-In) gelernt wird.

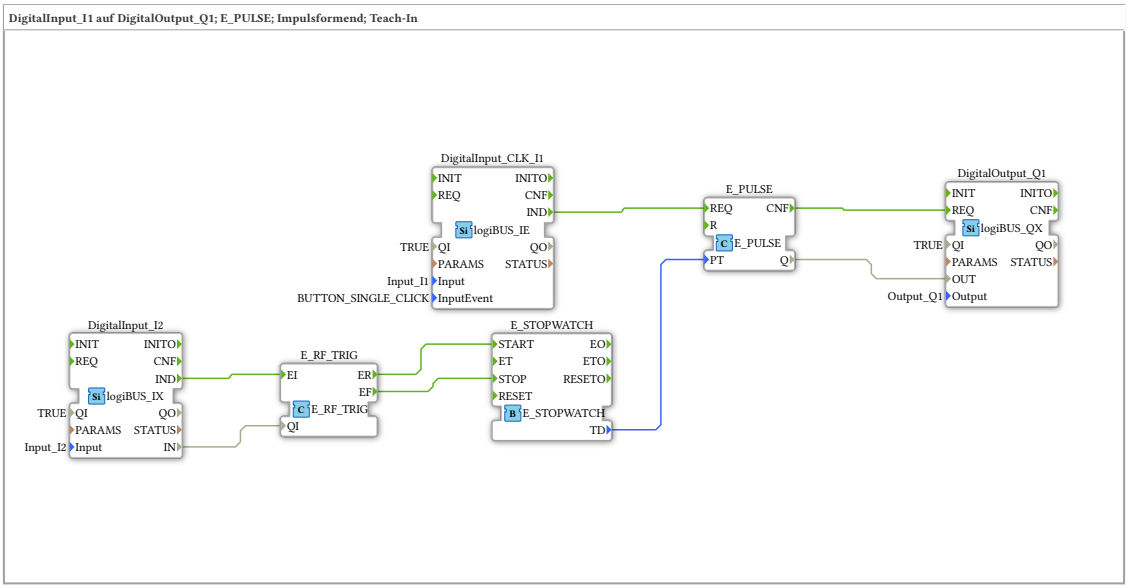
Ziel der Übung

Programmierung einer variablen Impulsdauer unter Verwendung des `E_STOPWATCH` Bausteins.

Beschreibung und Komponenten

Die Subapplikation `Uebung_020i.SUB` nutzt zwei Taster: Einen zum Ausführen und einen zum Lernen der Zeit.

Funktionsbausteine (FBs)



- **`E_STOPWATCH`**: Misst die Zeit zwischen einem Start- und einem Stopp-Ereignis.
- **`E_PULSE`**: Erzeugt den zeitgesteuerten Impuls.
- **`I2` (Lern-Taster)**: Ein normaler Pegel-Eingang (`IX`).
- **`I1` (Start-Taster)**: Ein Klick-Event-Eingang (`IE`).

Funktionsweise

1. **Lern-Modus**: Der Nutzer hält Taster `I2` gedrückt.
   - Beim Drücken (steigende Flanke) startet die Stoppuhr.
   - Beim Loslassen (fallende Flanke) stoppt die Stoppuhr.
   - Die gemessene Zeitdauer (`TD`) wird sofort an den Parameter `PT` des Pulsgebers übergeben.
2. **Arbeits-Modus**: Der Nutzer klickt kurz auf Taster `I1`.
   - Der `E_PULSE` wird getriggert.
   - Er schaltet den Ausgang für genau die Zeit an, die vorher mit Taster `I2` „vorgegeben“ wurde.

Anwendungsbeispiel

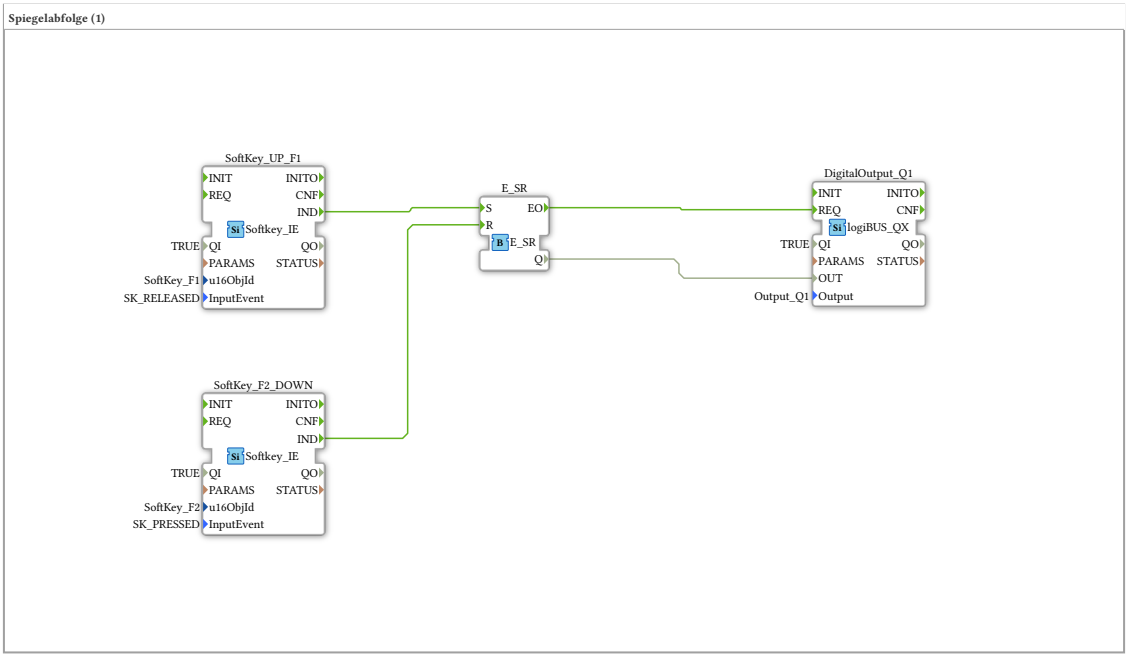
Zentralschmierung oder Bewässerung: Anstatt mühsam Sekundenwerte in ein Terminal einzutippen, drückt der Wartungstechniker einmalig so lange auf den Lern-Taster, wie er meint, dass der Vorgang dauern soll. Die Steuerung übernimmt diese Zeitspanne für alle zukünftigen automatischen Zyklen.

2.2.1.206 Uebung_021: Spiegelabfolge (1)

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_021`. Dies ist der Einstieg in die Ablaufsteuerung (Sequenzierung), simuliert am Beispiel eines Pneumatik-Zylinders.

🎧 Podcast



- Infineon MOTIX BTM9020/9021EP: Datenblatt-Analyse für Automotive – Robuster Motortreiber mit intelligenter Diagnose (HW vs. SPI)
- JBC Lötspitzen C470 vs. C245 vs. C210 vs. C115: Welche Spitze ist der Allrounder und wann brauchst du den Nano-Spezialisten?

Ziel der Übung

Realisierung einer einfachen Folgesteuerung: Ein Prozess wird gestartet und stoppt automatisch, sobald eine Endlage erreicht ist.

Beschreibung und Komponenten

Die Subapplikation `Uebung_021.SUB` nutzt zwei Softkeys, um die Bewegung eines Aktors (`Q1`) zu steuern.

Funktionsbausteine (FBs)

- **`SoftKey_UP_F1`**: Fungiert als **START-Taster**. Er ist auf `SK_RELEASED` konfiguriert.
- **`SoftKey_F2_DOWN`**: Simuliert den **Endlagenschalter**. Er reagiert sofort beim Drücken (`SK_PRESSED`).
- **`E_SR`**: Der Speicher für den Bewegungszustand.
- **`DigitalOutput_Q1`**: Der Ausgang für das Zylinderventil.

Funktionsweise

1. **Start**: Der Nutzer klickt auf **F1**. Das Ereignis setzt den Speicher `E_SR.S` ➡️ Der Ausgang `Q1` wird aktiv, der Zylinder fährt aus.
2. **Bewegung**: Der Zylinder bewegt sich physikalisch (oder in der Simulation) zur Endposition.
3. **Stopp**: Sobald der Zylinder die Endlage erreicht, wird (simuliert durch **F2**) der Reset-Eingang `E_SR.R` getriggert ➡️ Der Ausgang `Q1` wird deaktiviert, die Bewegung stoppt.

Anwendungsbeispiel

Einfacher Ausstoßer: In einer Fertigungslinie soll ein Paket per Knopfdruck vom Band geschoben werden. Der Bediener gibt den Startimpuls, der Zylinder fährt aus, schiebt das Paket weg und wird durch einen mechanischen Endschalter am Ende des Weges automatisch wieder abgeschaltet.

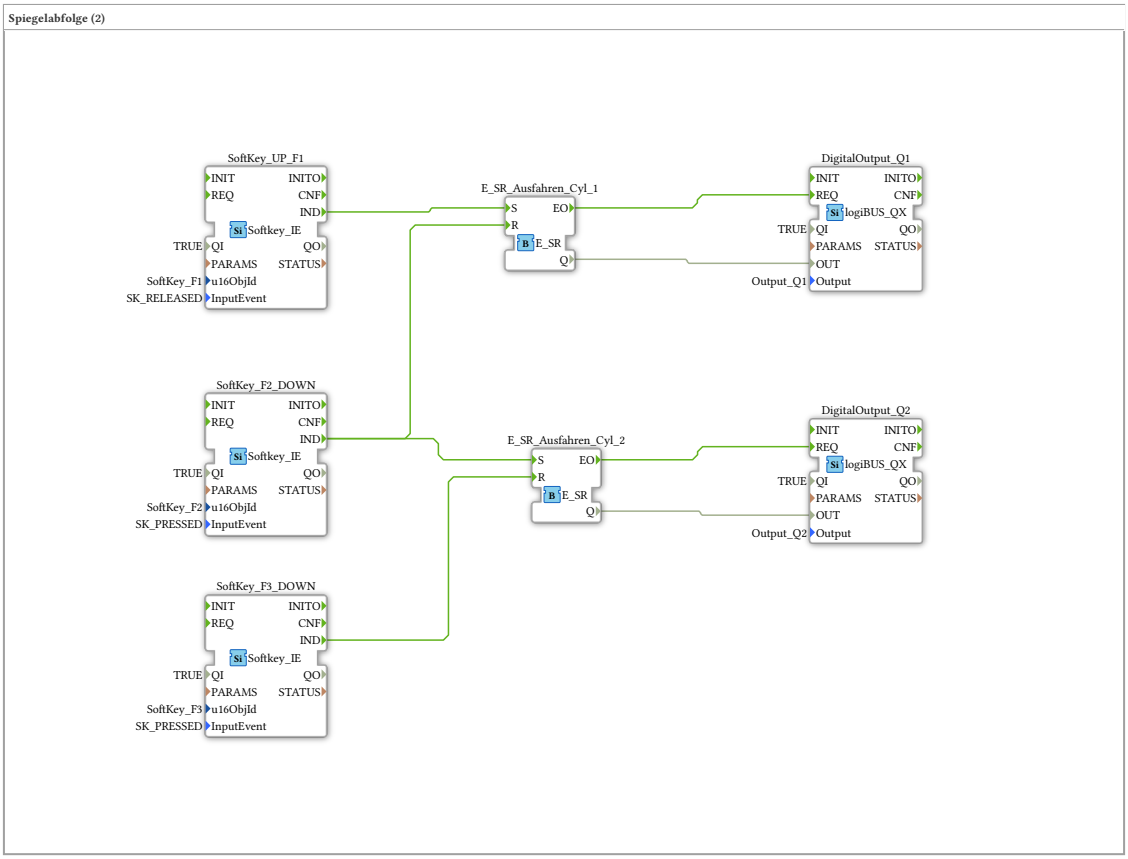
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

- 🌐 Interaktiver JBC-Lötspitzen Guide & Infografik auf ms-muc-docs.de

2.2.1.207 Uebung_022: Spiegelabfolge (2)

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_022`. Hier wird die Ablaufsteuerung auf zwei nacheinander folgende Schritte erweitert.

🎧 Podcast



- Als Landtechnik-Spezialist durch die Hölle: Wie Lanz-Wery Krieg, Besatzung und Hyperinflation überlebte – Einblicke in Original-Geschäftsberichte 1915-1922

Ziel der Übung

Erlernen der Ereignisverkettung. Das Ende eines Prozesses (Erreichen der Endlage) soll automatisch den nächsten Prozessschritt einleiten.

Beschreibung und Komponenten

In `Uebung_022.SUB` werden zwei Speicherglieder so verschaltet, dass eine Kaskade entsteht.

Funktionsbausteine (FBs)

- **`I1` (Start)**: Startet den gesamten Ablauf.
- **`I2` (Endlage 1)**: Beendet Schritt 1 und startet Schritt 2.
- **`I3` (Endlage 2)**: Beendet Schritt 2.
- **`Q1` & `Q2`**: Die Ausgänge für zwei Zylinder.

Funktionsweise

```
<EventConnections>
    <Connection Source="SoftKey_UP_F1.IND" Destination="E_SR_Cyl_1.S"/>
    <Connection Source="SoftKey_F2_DOWN.IND" Destination="E_SR_Cyl_1.R"/>
    <Connection Source="SoftKey_F2_DOWN.IND" Destination="E_SR_Cyl_2.S"/>
    <Connection Source="SoftKey_F3_DOWN.IND" Destination="E_SR_Cyl_2.R"/>
</EventConnections>
```

Der Ablauf:

1. Klick auf **F1** ➡️ Zylinder 1 fährt aus (`Q1`).
2. Zylinder 1 erreicht Endlage (**F2**) ➡️ `Q1` wird abgeschaltet **UND** zeitgleich wird Zylinder 2 gestartet (`Q2`).
3. Zylinder 2 erreicht seine Endlage (**F3**) ➡️ `Q2` wird abgeschaltet.

Anwendungsbeispiel

Zweistufige Paketübergabe: Zylinder 1 schiebt ein Paket aus einem Magazin auf einen Hubtisch. Sobald das Paket dort ankommt (Endschalter 1), stoppt Zylinder 1 und Zylinder 2 hebt den Tisch an.

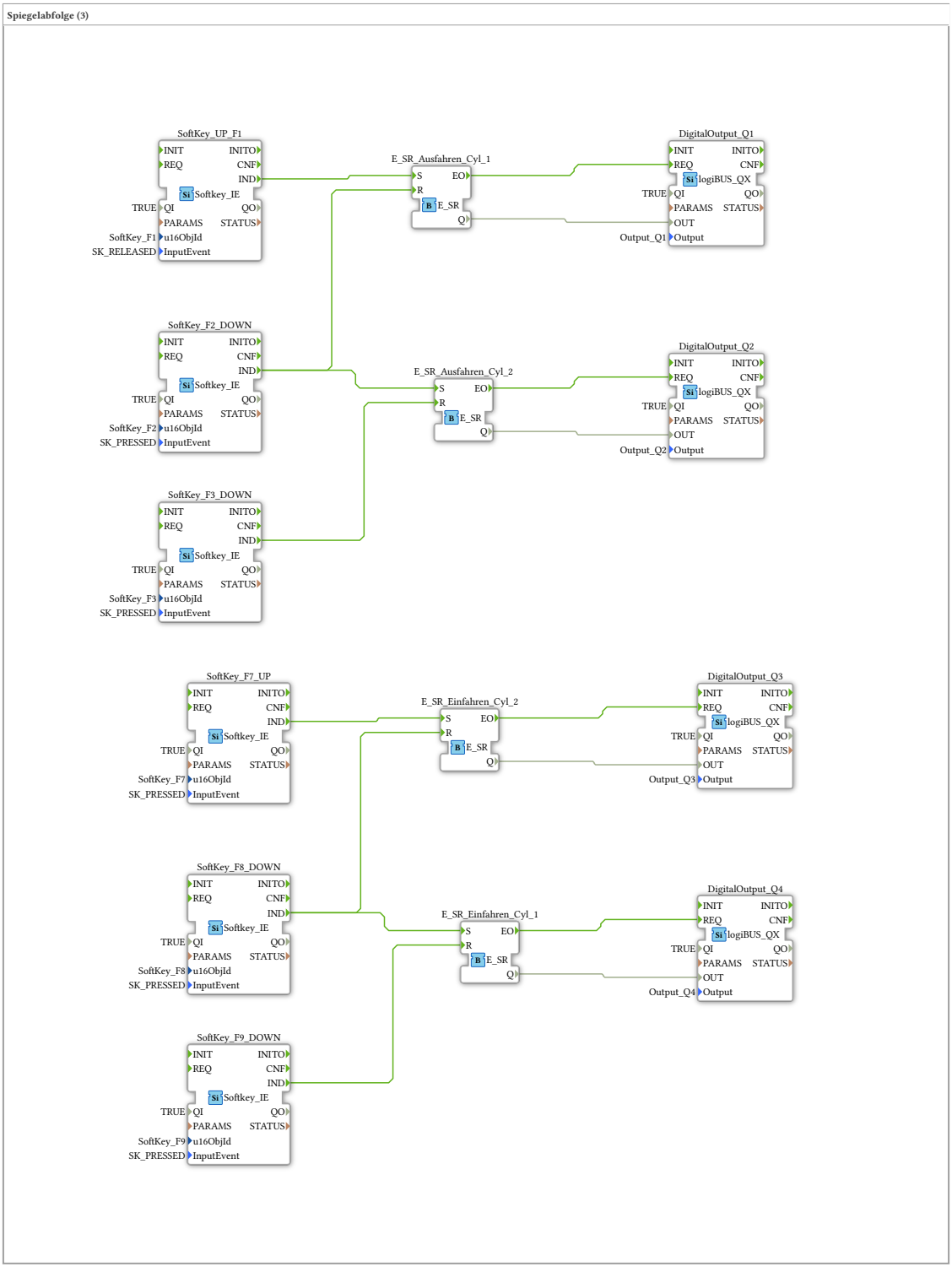
2.2.1.208 Uebung_023: Spiegelabfolge (3)



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_023`. Hier wird ein vollständiger Hin- und Rückweg für zwei Zylinder implementiert.

Übersicht

Diese Übung erweitert die Logik auf insgesamt vier Phasen unter Verwendung von sechs Softkeys:



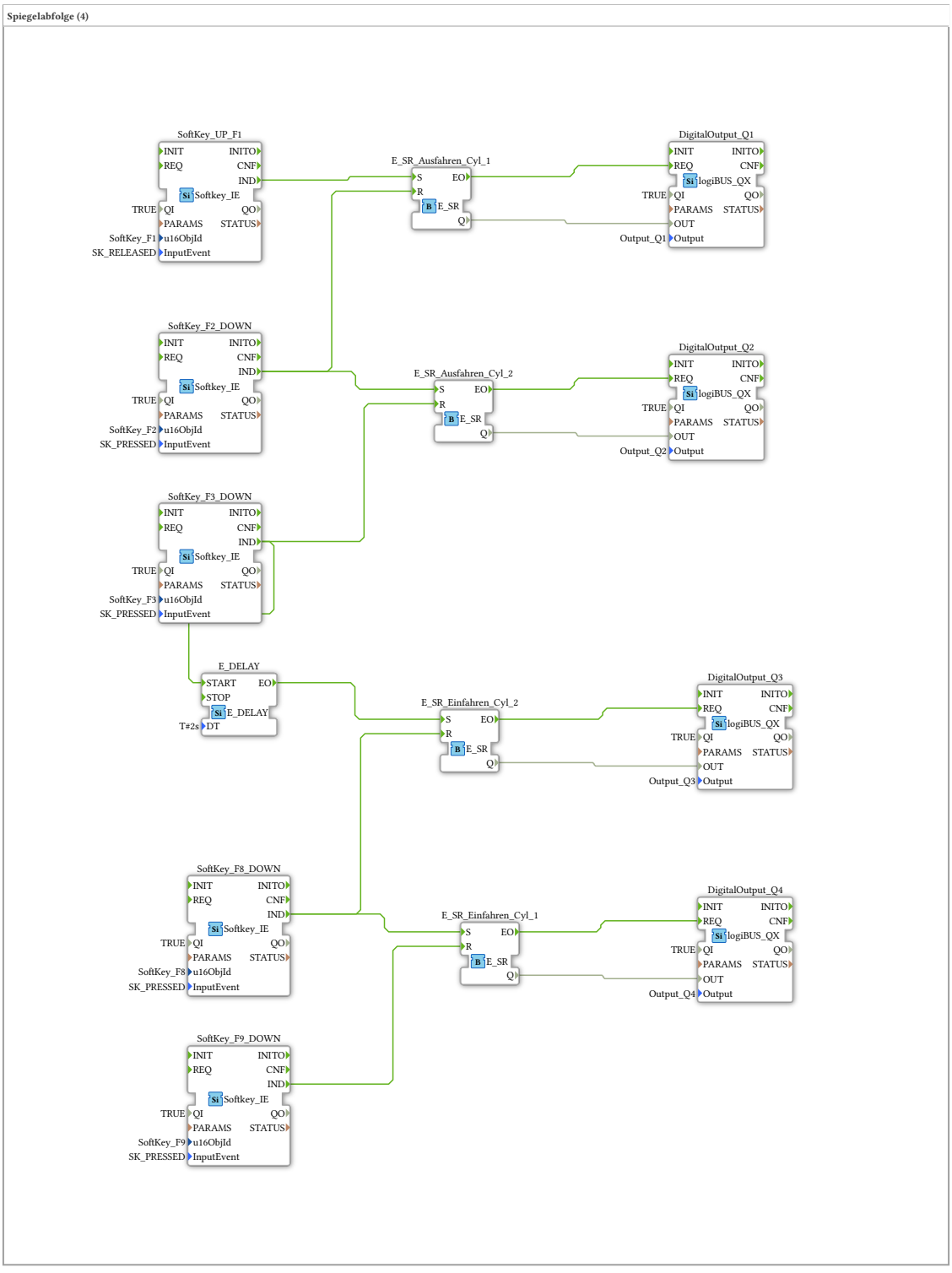
1. **Ausfahren**: `F1` (Start) ➡️ `Q1` an. Endlage erreicht über `F2`.
2. **Folgeschritt**: `F2` stoppt `Q1` und startet `Q2`. Endlage erreicht über `F3`.
3. **Einfahren**: Über separate Taster (`F7`, `F8`) wird die Rückhol-Sequenz eingeleitet (`Q3`, `Q4`).

Dies zeigt die Handhabung von komplexen, richtungsabhängigen Abläufen in einer flachen Ereignisstruktur.

2.2.1.209 Uebung_024: Spiegelabfolge (4)

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_024`. Hier wird eine zeitliche Pause in den automatischen Ablauf integriert.

🎧 Podcast



- Analyse der Novellierung der Meisterprüfungsverordnung im Land- und Baumaschinenmechatroniker-Handwerk: Ein Detaillierter Vergleich der Verordnungen von 2024 und 2001
- JBC Lötspitzen C470 vs. C245 vs. C210 vs. C115: Welche Spitze ist der Allrounder und wann brauchst du den Nano-Spezialisten?
- Strip-Till im Maisanbau: Wie Hochpräzision Wasser spart und den Boden schützt – Einblick in die Agrartechnik 2024

Ziel der Übung

Integration von Zeitgliedern (`E_DELAY`) in eine Ereigniskette.

Funktionsweise

Im Vergleich zu Übung 023 wird zwischen zwei Schritten ein Verzögerungs-Baustein eingefügt. Wenn Zylinder 2 seine Endlage erreicht (`F3`), wird nicht sofort der nächste Schritt ausgelöst, sondern der Eingang `E_DELAY.START` getriggert. Erst nach Ablauf der Zeit `DT` (hier 2 Sekunden) feuert das `EO`-Event und setzt die Sequenz fort (z.B. Start des Einfahrens).

Anwendungsbeispiel

Pressvorgang: Ein Zylinder fährt aus und drückt zwei Bauteile zusammen. Sobald die Endlage erreicht ist, muss der Druck für 2 Sekunden gehalten werden (Wartezeit), bevor der Zylinder wieder einfährt und das Werkstück freigibt.

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

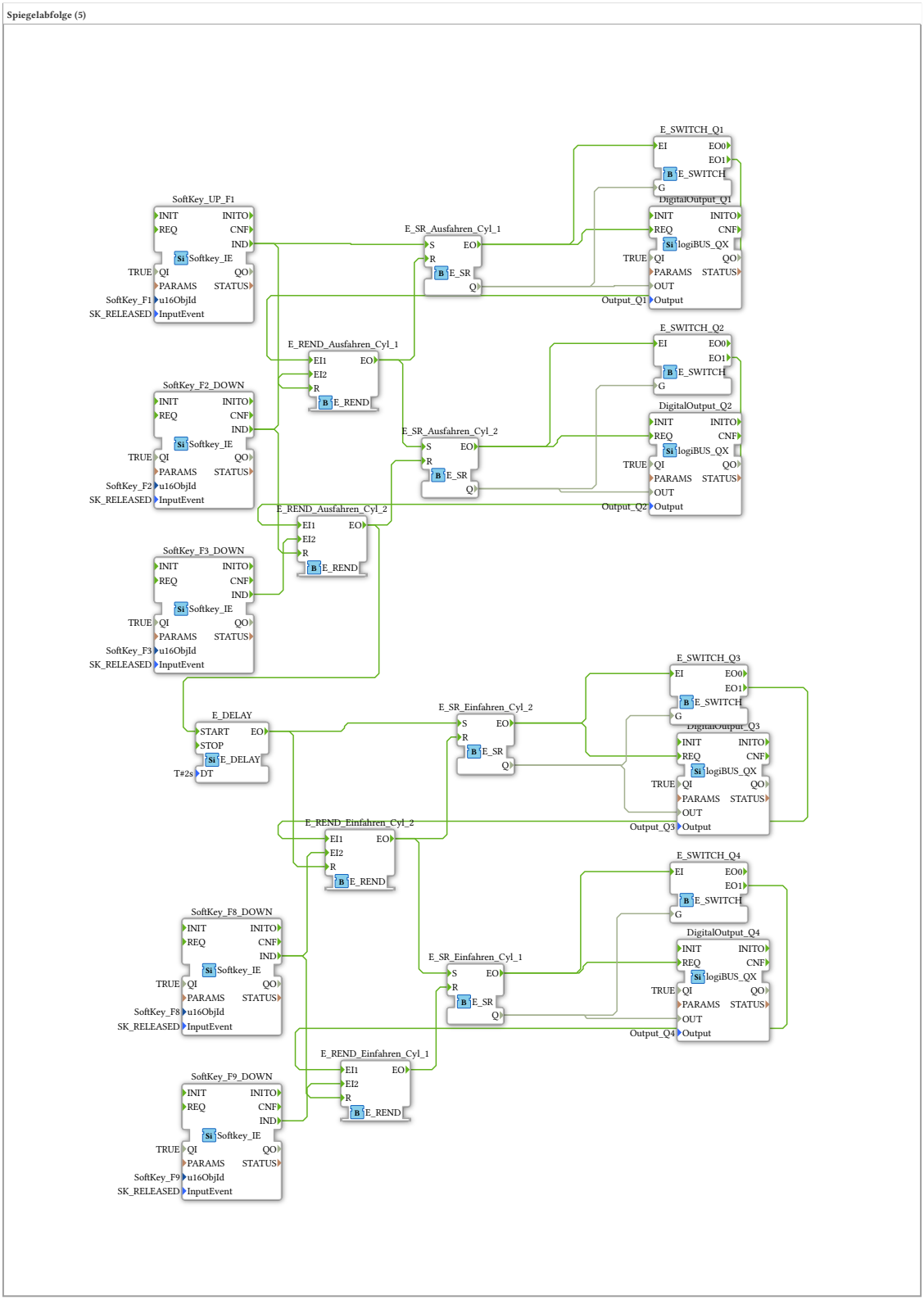
- 🌐 Interaktiver JBC-Lötspitzen Guide & Infografik auf ms-muc-docs.de

2.2.1.210 Uebung_025: Spiegelabfolge (5)



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_025`. Hier wird die Ablaufsteuerung durch Rendezvous-Bausteine abgesichert.

📺 Video



- Löten im Jahr 2025

🎧 Podcast

- ETFA 2025: Plug and Produce – Wie IEC 61499 die Fabrikautomation revolutioniert
- IEC 61499: Sprung in die Industrie – ETFA 2025 und die Zukunft der Automatisierung

- Industrial Revolution Reloaded: Unpacking Plug & Produce, Data Privacy, and ETFA 2025
- Revolutionen der Industrie: Von Dampfmaschine bis KI – Ein tiefer Einblick in 250 Jahre Automatisierung
- Das Kettenmonster erwacht: Lanz Bulldog Raupe – Die faszinierende Wiederbelebung des 10-Liter-Glühkopf-Arbeitstiers nach 25 Jahren Stillstand

---

Ziel der Übung

Verwendung von `E_REND` zur Absicherung von Übergängen. Es soll sichergestellt werden, dass ein Folgeschritt nur dann ausgelöst wird, wenn sowohl das Hardware-Feedback (Endlage) als auch das logische Software-Event (Bereitschaft) vorliegen.

Funktionsweise

Die Übung nutzt für jeden Übergang einen `E_REND` Baustein. Zusätzlich werden `E_SWITCH` Bausteine zur Plausibilitätsprüfung eingesetzt. Ein Ereignis wird nur dann als gültige Endlage akzeptiert, wenn der zugehörige Ausgang (`Q`) der Steuerung zu diesem Zeitpunkt auch tatsächlich aktiv ist (Rückkopplung der Daten an das Gate der Weiche). Dies verhindert Fehlsteuerungen durch defekte oder hängende Sensoren.

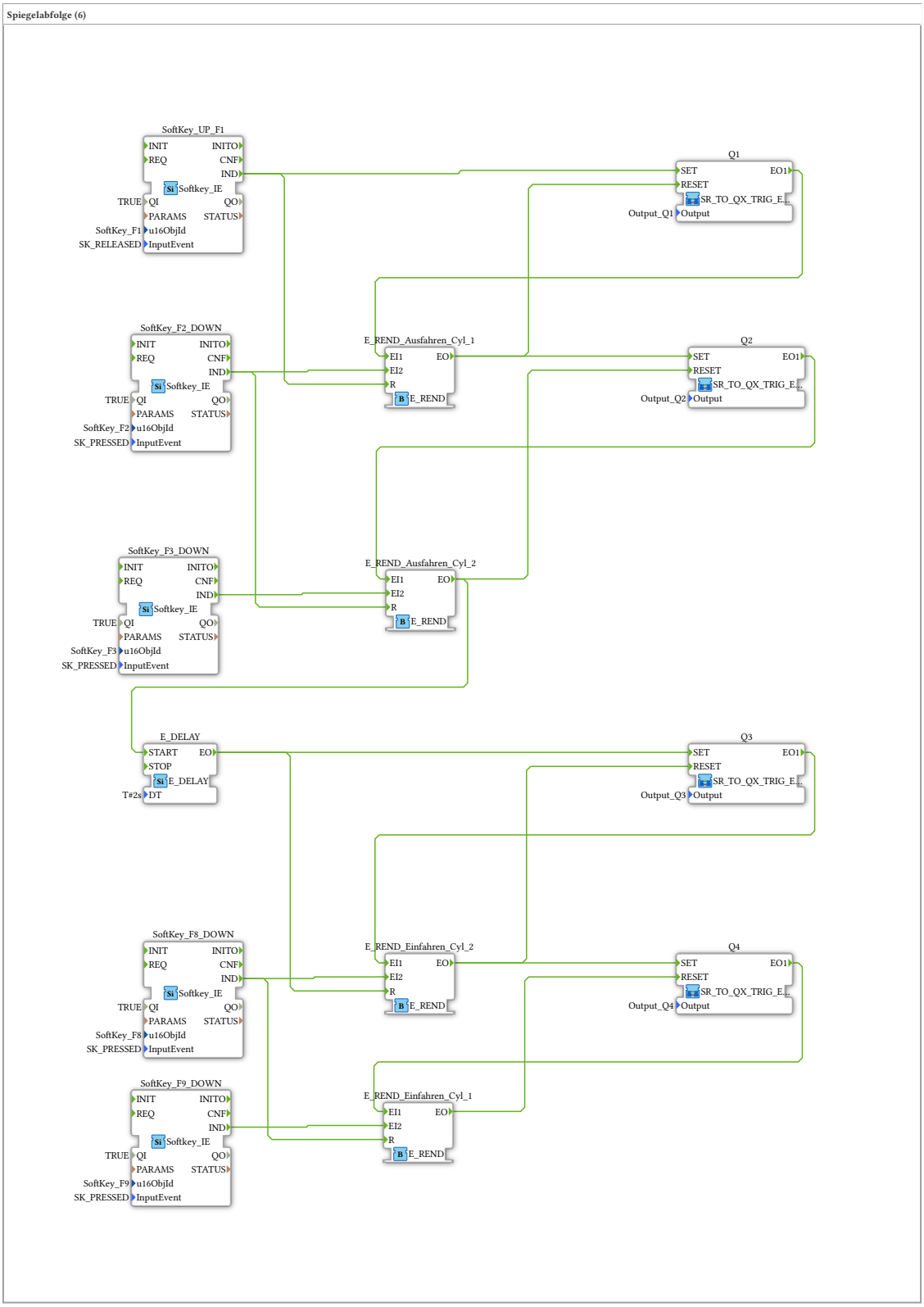
2.2.1.211 Uebung_026: Spiegelabfolge (6)



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_026`.

Übersicht

In dieser Übung wird die komplexe Ablauflogik aus Übung 025 beibehalten, jedoch wird die Ansteuerung der Hardware-Ausgänge in eine typisierte Sub-Applikation `Uebung_026_sub` ausgelagert. Jede Instanz dieser Sub-App (`Q1` bis `Q4`) kapselt einen SR-Speicher und den Hardware-Ausgangsbaustein. Das Hauptdiagramm wird dadurch wesentlich übersichtlicher, da nur noch die Ereigniskette zwischen den Phasen (Rendezvous und Delays) sichtbar ist, während die „Leistungsebene“ im Hintergrund arbeitet.



2.2.1.212 Uebung_026_sub

Hinweis

Diese Übung existiert nicht mehr als eigene Übung. Der Baustein wurde in die Bibliothek verschoben und heißt jetzt SR_TO_QX_TRIG_EO. Beschreibung, Schnittstelle und Netzwerk stehen dort.

2.2.1.213 Uebung_028: Analog-Eingang

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_028`. Hier verlassen wir die digitale Welt (An/Aus) und erfassen kontinuierliche Messwerte (Analogsignale).

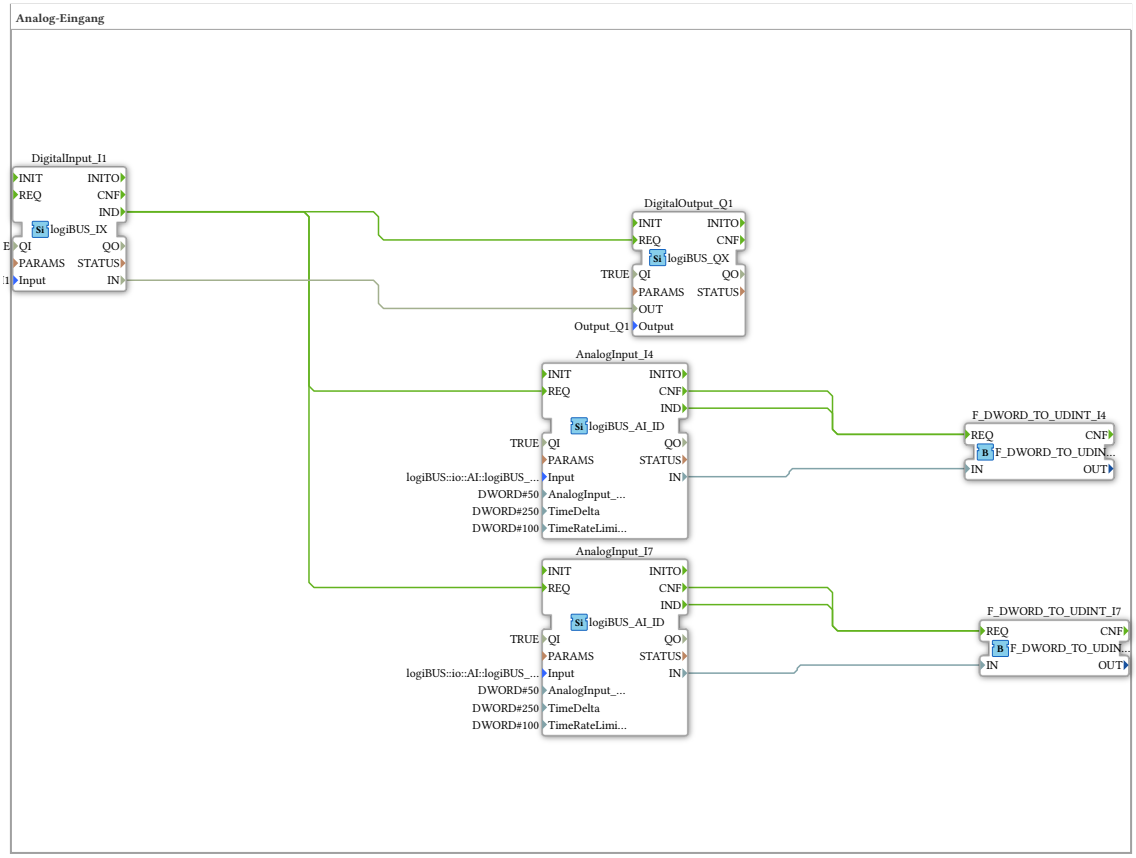
Ziel der Übung

Verwendung des Bausteins `logiBUS_AI_ID`. Es wird demonstriert, wie analoge Spannungswerte (z.B. von einem Potentiometer oder Sensor) eingelesen, gefiltert (Hysterese) und konvertiert werden.

Beschreibung und Komponenten

Die Subapplikation `Uebung_028.SUB` liest zwei Analogkanäle der Hardware ein.

Funktionsbausteine (FBs)



- **`AnalogInput_I4` & `I7`**: Typ `logiBUS_AI_ID`. Diese Bausteine repräsentieren die analogen Hardware-Eingänge. Sie wandeln die elektrische Spannung in einen numerischen Digitalwert um.
- **Parameter `AnalogInput_hysteresis`**: Bestimmt, um wie viel sich der Wert ändern muss, bevor ein neues Ereignis (`IND`) gefeuert wird (hier 50 Einheiten). Dies unterdrückt Rauschen.
- **`F_DWORD_TO_UDINT`**: Konvertiert den Rohwert in einen Ganzzahl-Datentyp zur weiteren Verarbeitung.

Funktionsweise

Der Analogbaustein bietet zwei Möglichkeiten der Abfrage:

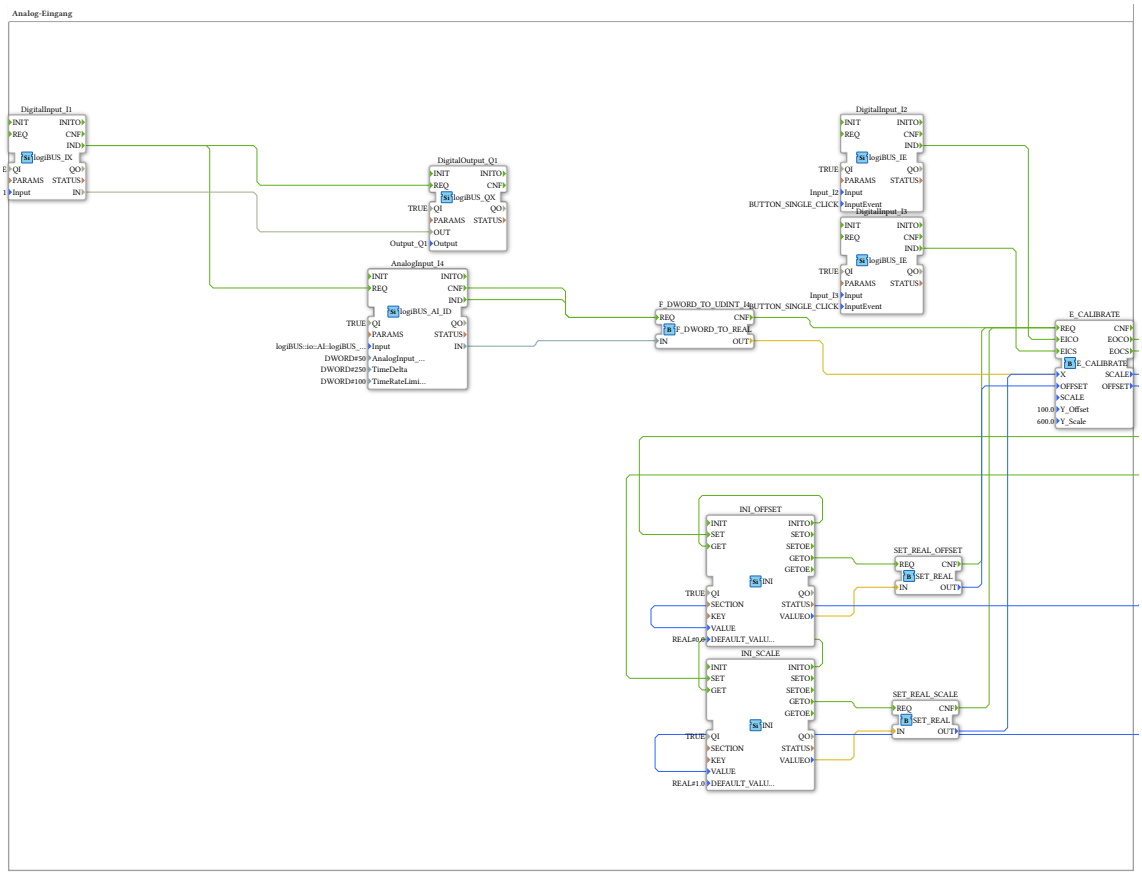
1. **Ereignisgesteuert**: Sobald sich die Eingangsspannung signifikant ändert (außerhalb der Hysterese), sendet der Baustein automatisch ein `IND`-Event.

2. **Manuell (Polling)**: In dieser Übung triggert zusätzlich der digitale Taster `I1` den `REQ`-Eingang der Analog-Bausteine. Dies erzwingt eine sofortige Aktualisierung der Werte, egal ob sie sich geändert haben oder nicht.

Anwendungsbeispiel

Tankinhalts-Anzeige: Ein Schwimmersensor im Tank liefert eine analoge Spannung. Die Steuerung liest diesen Wert ein. Durch die Hysterese wird verhindert, dass die Anzeige bei leichtem Schwanken des Kraftstoffs ständig flackert. Der Nutzer kann jederzeit einen Knopf am Bedienpult drücken, um den absolut aktuellen Wert sofort abzufragen.

2.2.1.214 Uebung_028a: Analog-Eingang



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert die Verarbeitung eines analogen Eingangssignals mit Kalibrierung. Es werden digitale Taster zur Kalibrierung von Offset und Skalierung verwendet, und die ermittelten Kalibrierungsparameter werden nichtflüchtig gespeichert. Die Übung zeigt den Umgang mit analogen Eingangsbausteinen, Typkonvertierung, Kalibrierungsfunktionen und Speicherbausteinen in der 4diac-IDE.

Verwendete Funktionsbausteine (FBs)

- **DigitalOutput_Q1** (logiBUS_QX): Digitaler Ausgang Q1.
- **DigitalInput_I1** (logiBUS_IX): Digitaler Eingang I1.
- **AnalogInput_I4** (logiBUS_AI_ID): Analoger Eingang I4.
  - Parameter: AnalogInput_hysteresis = 50, TimeDelta = 250, TimeRateLimit = 100.
- **F_DWORD_TO_UDINT_I4** (F_DWORD_TO_REAL): Konvertierung eines DWORD-Wertes nach REAL.
- **CALIBRATE** (E_CALIBRATE): Kalibrierungsbaustein.
  - Parameter: Y_Offset = 100.0, Y_Scale = 600.0.

- **DigitalInput_I2** (logiBUS_IE): Digitaler Eingang I2 mit Ereignis BUTTON_SINGLE_CLICK – dient als Taster zur Offset-Kalibrierung.
- **DigitalInput_I3** (logiBUS_IE): Digitaler Eingang I3 mit Ereignis BUTTON_SINGLE_CLICK – dient als Taster zur Skalierungs-Kalibrierung.
- **INI_OFFSET** (INI): Speicherbaustein für den Offset-Wert.
  - Parameter: DEFAULT_VALUE = REAL#0.0.
- **SET_REAL_OFFSET** (SET_REAL): Stellt den gespeicherten Offset-Wert als REAL bereit (initial 0.0).
- **INI_SCALE** (INI): Speicherbaustein für den Skalierungs-Wert.
  - Parameter: DEFAULT_VALUE = REAL#1.0.
- **SET_REAL_SCALE** (SET_REAL): Stellt den gespeicherten Skalierungs-Wert als REAL bereit (initial 1.0).

Programmablauf und Verbindungen

Der Ablauf wird durch Ereignisse gesteuert:

1. **DigitalInput_I1** sendet bei Betätigung ein `IND`-Ereignis. Dieses triggert gleichzeitig den digitalen Ausgang **DigitalOutput_Q1** (über `REQ`) und startet den analogen Eingang **AnalogInput_I4** (über `REQ`).
2. **AnalogInput_I4** erfasst einen analogen Wert und gibt ihn als DWORD an seinem Ausgang `IN` aus. Gleichzeitig wird ein `IND`-Ereignis gesendet, das den Konvertierungsbaustein **F_DWORD_TO_UDINT_I4** aktiviert.
3. **F_DWORD_TO_UDINT_I4** wandelt den DWORD-Wert nach REAL um und übergibt das Ergebnis an den Kalibrierungsbaustein **CALIBRATE** über dessen Eingang `X`.
4. Der Benutzer kann die Kalibrierung manuell auslösen:
   - **DigitalInput_I2** (Taster) sendet ein `IND`-Ereignis an `EICO` von **CALIBRATE** ⟶ löst die Offset-Kalibrierung aus.
   - **DigitalInput_I3** (Taster) sendet ein `IND`-Ereignis an `EICS` von **CALIBRATE** ⟶ löst die Skalierungs-Kalibrierung aus.
5. **CALIBRATE** berechnet aus dem Rohwert und den aktuellen Kalibrierparametern (Offset und Skalierung) den korrigierten Wert. Die neuen Parameter werden an den Ausgängen `OFFSET` und `SCALE` ausgegeben.
6. Diese neuen Parameter werden über Datenverbindungen in die Speicherbausteine **INI_OFFSET** und **INI_SCALE** geschrieben (Ereignis `SET` wird von **CALIBRATE** über `EOCO` bzw. `EOCS` gesendet).
7. Die gespeicherten Werte werden nach dem Initialisieren (`INITO` ⟶ `GET`) über **SET_REAL_OFFSET** und **SET_REAL_SCALE** zurück an **CALIBRATE** geführt, sodass die Kalibrierung dauerhaft erhalten bleibt.

Die Datenflüsse verbinden:

- `AnalogInput_I4.IN` ⟶ `F_DWORD_TO_UDINT_I4.IN`
- `F_DWORD_TO_UDINT_I4.OUT` ⟶ `CALIBRATE.X`
- `INI_OFFSET.VALUEO` ⟶ `SET_REAL_OFFSET.IN` ⟶ `SET_REAL_OFFSET.OUT` ⟶ `CALIBRATE.OFFSET`
- `INI_SCALE.VALUEO` ⟶ `SET_REAL_SCALE.IN` ⟶ `SET_REAL_SCALE.OUT` ⟶ `CALIBRATE.SCALE`
- Zurückschreiben: `CALIBRATE.OFFSET` ⟶ `INI_OFFSET.VALUE`, `CALIBRATE.SCALE` ⟶ `INI_SCALE.VALUE`

Zusammenfassung

Die Übung vermittelt den Umgang mit analogen Eingängen, deren Typkonvertierung sowie die Implementierung einer benutzergesteuerten Kalibrierung. Die Kalibrierungsparameter (Offset und Skalierung) werden in nichtflüchtigen Speichern gehalten und können über Taster angepasst werden. Der Beispielcode zeigt, wie Ereignis- und Datenflüsse in einer SubApp strukturiert werden können, um eine robuste und wiederholgenaue Analogwerterfassung zu realisieren.

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

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

2.2.1.215 Uebung_029: LED_DO Blinkende LED



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_029`. Hier wird ein spezialisierter Baustein zur Ansteuerung von Status-LEDs vorgestellt, der das Blinken hardwarenah übernimmt.

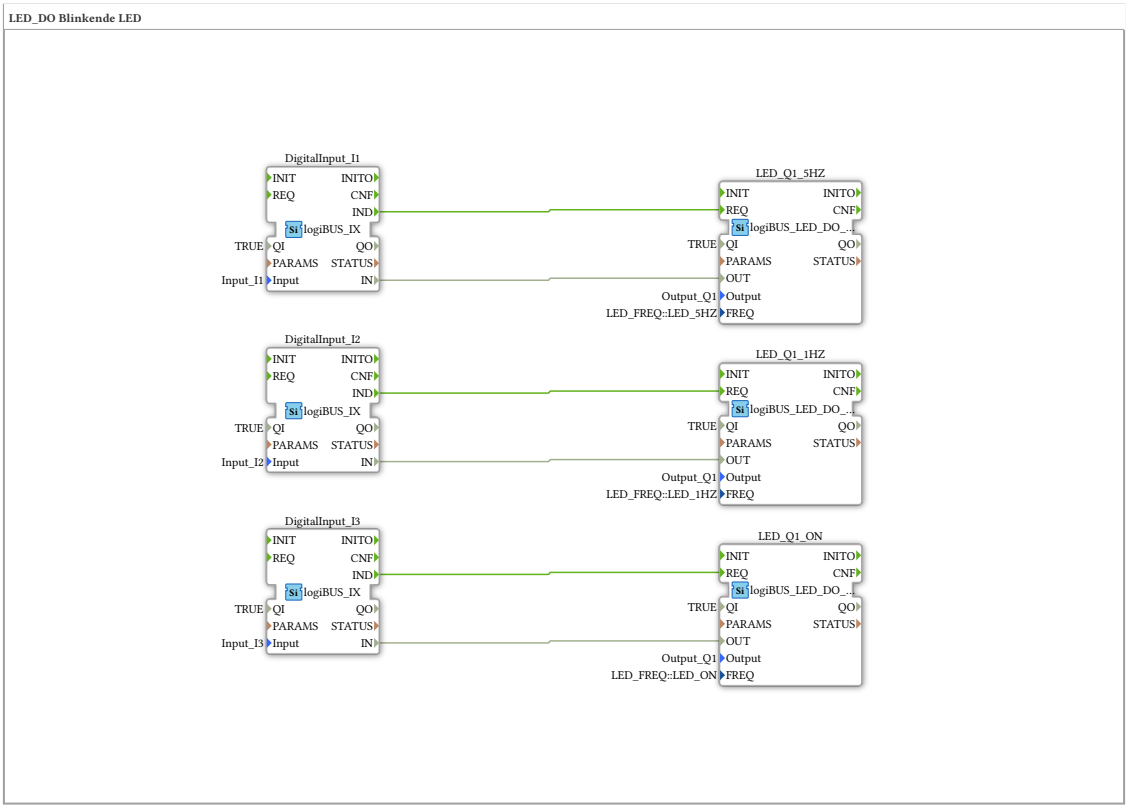
Ziel der Übung

Nutzung des Bausteins `logiBUS_LED_DO_QX`. Es wird gezeigt, wie man eine LED in verschiedenen Modi (Dauerlicht, langsames Blinken, schnelles Blinken) betreibt, ohne dafür komplexe Software-Timer oder Blink-Logiken (wie in Übung 007) programmieren zu müssen.

Beschreibung und Komponenten

In `Uebung_029.SUB` werden drei Taster genutzt, um eine einzige LED (`Q1`) in drei verschiedenen Modi anzusteuern.

Funktionsbausteine (FBs)



- **`logiBUS_LED_DO_QX`**: Ein spezialisierter Ausgangsbaustein. Er besitzt den Parameter `FREQ` (Frequenz).
- **Parameter `FREQ`**:
  - `LED_ON`: Dauerhaftes Leuchten.
  - `LED_1HZ`: Langsames Blinken (1 mal pro Sekunde).
  - `LED_5HZ`: Schnelles Blinken (5 mal pro Sekunde).

Funktionsweise

Obwohl alle drei Bausteine im Diagramm auf denselben physischen Ausgang `Output_Q1` verweisen, unterscheiden sie sich in ihrer Konfiguration:

- Druck auf **Taster I1** ➡️ Triggert den 5Hz-Baustein. Die LED blitzt sehr schnell.
- Druck auf **Taster I2** ➡️ Triggert den 1Hz-Baustein. Die LED blinkt ruhig.
- Druck auf **Taster I3** ➡️ Triggert den ON-Baustein. Die LED leuchtet konstant.

Die Blink-Frequenz wird dabei direkt vom Hardware-Treiber der Steuerung generiert, was den Prozessor entlastet.

Anwendungsbeispiel

Zustands-Signalisierung einer Maschine:

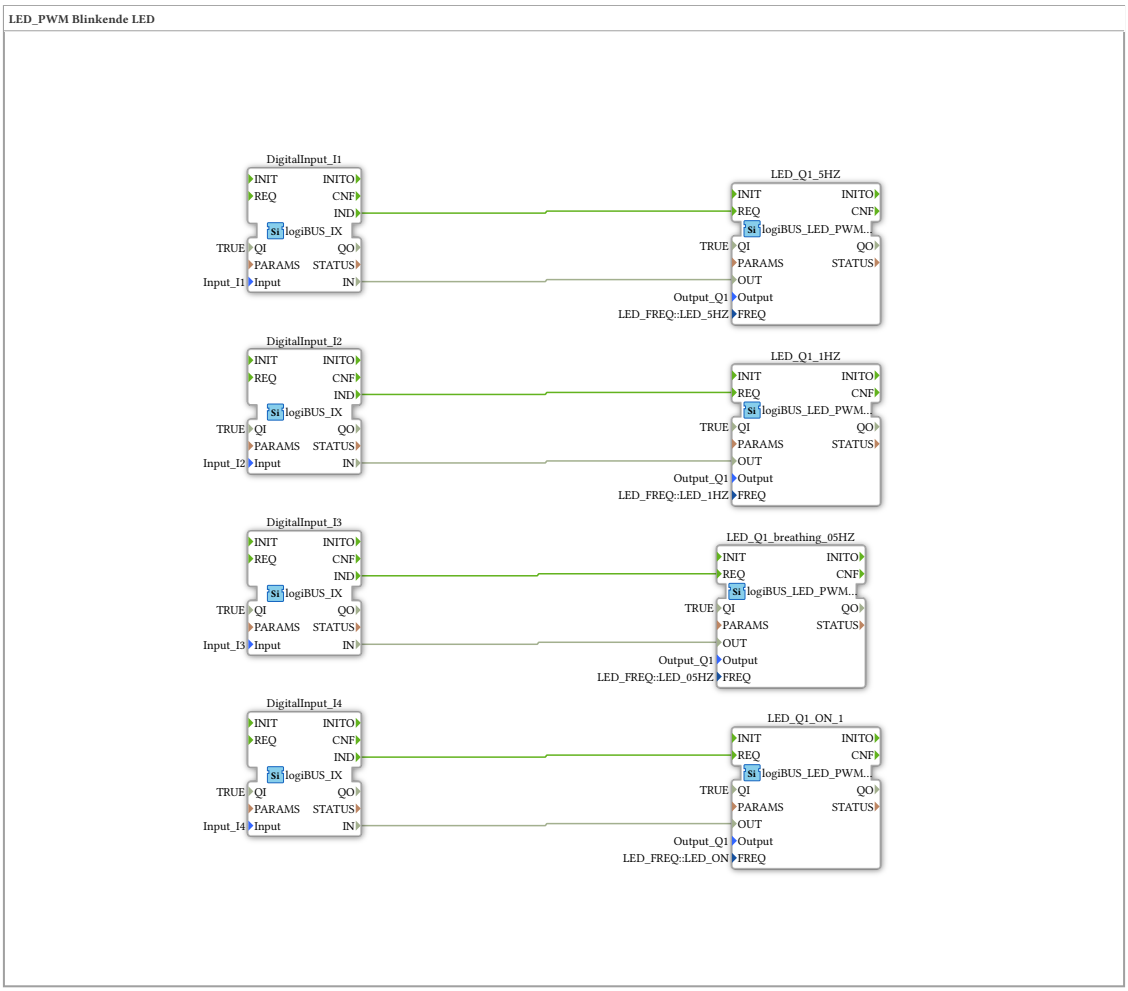
- **LED An**: Maschine ist bereit.
- **LED 1Hz**: Maschine arbeitet (Automatikbetrieb).
- **LED 5Hz**: Warnung oder Störung (Aufmerksamkeit erforderlich). Dies ermöglicht eine intuitive Kommunikation mit dem Bediener über eine einzige Kontrollleuchte.

2.2.1.216 Uebung_030: LED_PWM Blinkende LED

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_030`. Hier werden die erweiterten Fähigkeiten der LED-Ansteuerung mittels Pulsweitenmodulation (PWM) demonstriert.

🎧 Podcast



- 3000 Watt Lüge Die TVS Diode entschlüsselt
- Der BTS7030-2EPA intelligenter Auto Stromwächter
- Der Intelligente Leistungsschalter: Wie der Infineon BTS7030 Relais und Sicherungen im Auto ersetzt
- Infineon BTS7030-2EPA: Intelligenter High-Side Leistungsschalter

---

Ziel der Übung

Verwendung des Bausteins `logiBUS_LED_PWM_QX`. Es wird gezeigt, wie man weiche Lichteffekte (Pulsieren / „Breathing“) realisiert, indem man die Helligkeit der LED über die Hardware-PWM steuert.

Beschreibung und Komponenten

In `Uebung_030.SUB` werden vier Taster genutzt, um verschiedene PWM-Effekte auf einer LED (`Q1`) auszulösen.

Funktionsbausteine (FBs)

- **`logiBUS_LED_PWM_QX`**: Dieser Baustein nutzt die PWM-Hardware, um nicht nur AN/AUS, sondern auch Helligkeitsverläufe darzustellen.
- **Parameter `FREQ`**:
  - `LED_05HZ`: Ein sehr langsamer „Breathing“-Effekt (Pulsieren der Helligkeit).
  - `LED_1HZ` & `LED_5HZ`: Klassische Blinkfrequenzen.
  - `LED_ON`: Konstante Helligkeit (100%).

Funktionsweise

Jeder Taster aktiviert eine andere Instanz des PWM-Bausteins, die alle auf denselben physikalischen Ausgang `Output_Q1` wirken.

- **Taster I3** ➡️ Aktiviert den 0,5 Hz Breathing-Effekt. Die LED wird langsam heller und wieder dunkler.
- **Taster I1 & I2** ➡️ Aktivieren schnelles oder langsames Blinken.
- **Taster I4** ➡️ Schaltet die LED auf Dauerlicht.

Anwendungsbeispiel

Hochwertige Statusanzeige: Anstatt eines harten Blinkens wird ein sanftes Pulsieren der LED genutzt, um einen „Standby“-Zustand oder einen laufenden, unkritischen Hintergrundprozess anzuzeigen. Dies wirkt moderner und weniger störend für den Bediener.

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

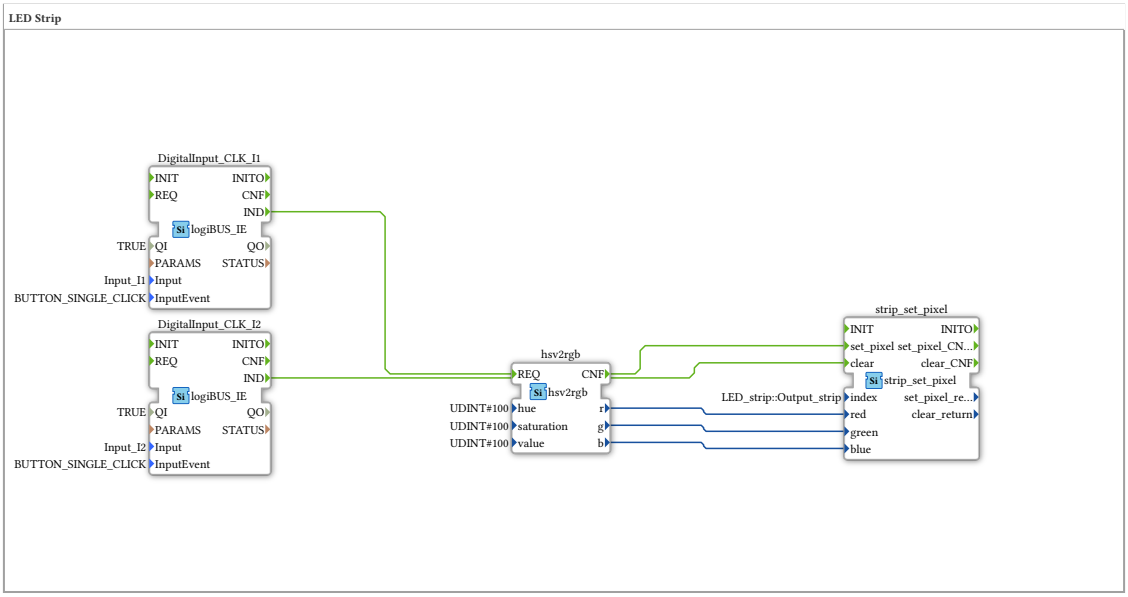
- 🌐 Diode & Halbleitergrundlagen auf ms-muc-docs.de
- 🌐 Das PWM-Signal & Infografik auf ms-muc-docs.de

2.2.1.217 Uebung_031: LED Strip



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_031`. Hier steuern wir adressierbare RGB-LEDs (z.B. WS2812) über das komfortable HSV-Farbmodell an.

🎧 Podcast



- Die drei Timer der DIN EN 61131-3 entschlüsselt – TP, TON & TOF präzise erklärt
- DIN EN 61131-3 vs. 61499-1: Dein Wegweiser durch die Normen der Industrieautomatisierung
- DIN EN 61131-3: Das Herz der Land- und Baumaschinen-Mechatronik und der Sprung in die Zukunft mit Ob
- FB_TOF und E_TOF: Verzögerungstimer in IEC 61131-3 und 61499
- IEC 61499 vs. 61131: Brauchen wir einen neuen Standard für IIoT? Analyse einer hitzigen Debatte um Verteilte Intelligenz

---

Ziel der Übung

Verwendung der RGB-Bibliothek für den ESP32. Es wird demonstriert, wie man Farben nicht über Rot-Grün-Blau-Werte (RGB), sondern über Farbwert (Hue), Sättigung (Saturation) und Helligkeit (Value) definiert und an einen LED-Streifen sendet.

Beschreibung und Komponenten

Die Subapplikation `Uebung_031.SUB` nutzt einen Konvertierungsbaustein und einen Streifen-Treiber.

Funktionsbausteine (FBs)

- **`hsv2rgb`**: Rechnet die intuitiven HSV-Werte in die von der Hardware benötigten RGB-Werte um.
- **`strip_set_pixel`**: Überträgt die Farbwerte an eine spezifische LED im Streifen.
- **`I1` (Set)**: Klick löst das Setzen der Farbe aus.
- **`I2` (Clear)**: Klick löscht die Anzeige (LED aus).

Funktionsweise

1. Der Nutzer klickt auf **I1**. Das Event triggert die Konvertierung.
2. Der `hsv2rgb` Baustein nimmt die voreingestellten Werte (z.B. Hue=100) und liefert die Anteile für Rot, Grün und Blau.
3. Das `CNF`-Event des Konverters startet den Hardware-Transfer über `strip_set_pixel`.
4. Die erste LED am Streifen leuchtet in der gewählten Farbe.

Anwendungsbeispiel

Individuelle Design-Beleuchtung: In einer Kabine soll die Ambiente-Beleuchtung einstellbar sein. Über ein Drehrad (Poti) wird der `Hue`-Wert verändert. Das Programm rechnet dies permanent um, sodass der Fahrer stufenlos durch den gesamten Regenbogen navigieren kann.

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

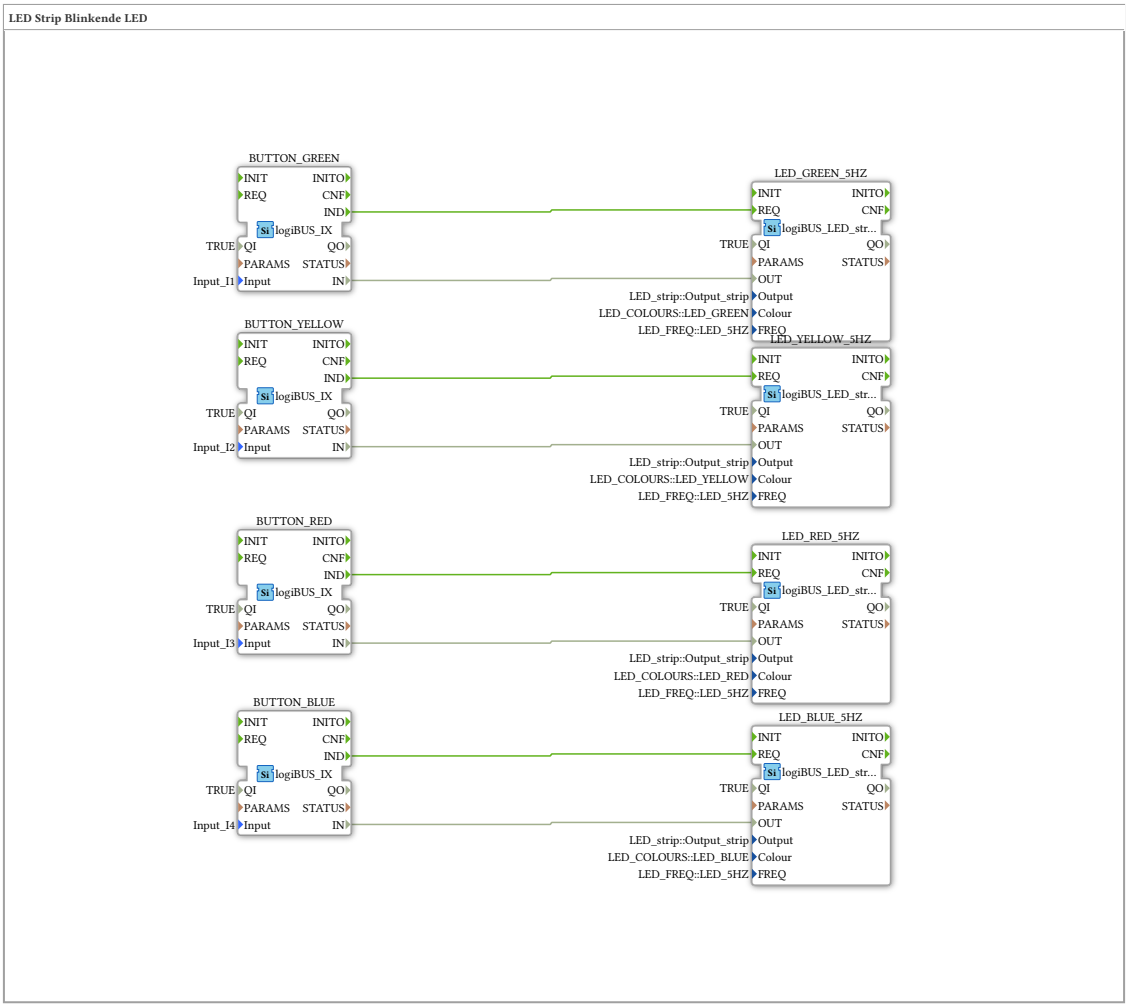
- 🌐 ESP32 & ESP32-S3 DevKit auf ms-muc-docs.de

2.2.1.218 Uebung_032: LED Strip Blinkende LED



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_032`. Hier werden vorkonfigurierte Farbbausteine für LED-Streifen genutzt.

📺 Video



- Das ESP32-S3-DevKitC-1

🎧 Podcast

- ESP32 als Industrie-SPS: Revolution mit Eclipse 4diac und logiBUS®
- ESP32 as PLC: Democratizing Industrial Automation with Eclipse 4diac
- ESP32 wird industrielle SPS für Landmaschinen
- ESP32-S3 Entwicklungsplatinen ESP32-S3-DevKitC-1
- ESP32-S3 im Detail: Dual-Core, 32MB Power und CAN-Bus für Land- und Baumaschinen-Mechatronik

---

Ziel der Übung

Verwendung des Bausteins `logiBUS_LED_strip_QX`. Dies ist ein High-Level Baustein, der Farbe, Frequenz und Hardware-Anbindung für RGB-Streifen in einem Block vereint.

Beschreibung und Komponenten

In `Uebung_032.SUB` werden vier verschiedene Farben (Grün, Gelb, Rot, Blau) auf vier Taster gemappt.

Funktionsbausteine (FBs)

- **`logiBUS_LED_strip_QX`**: Kombi-Baustein für RGB-Streifen.
- **Parameter**:
  - `Colour`: Auswahl aus einer Palette (z.B. `LED_RED`).
  - `FREQ`: Blinkfrequenz (hier einheitlich 5 Hz).

Funktionsweise

Jeder Taster aktiviert „seine“ Farbe auf dem Streifen. Da alle Bausteine auf den Parameter `Output_strip` (Kanal 0) konfiguriert sind, überschreiben sie sich gegenseitig.

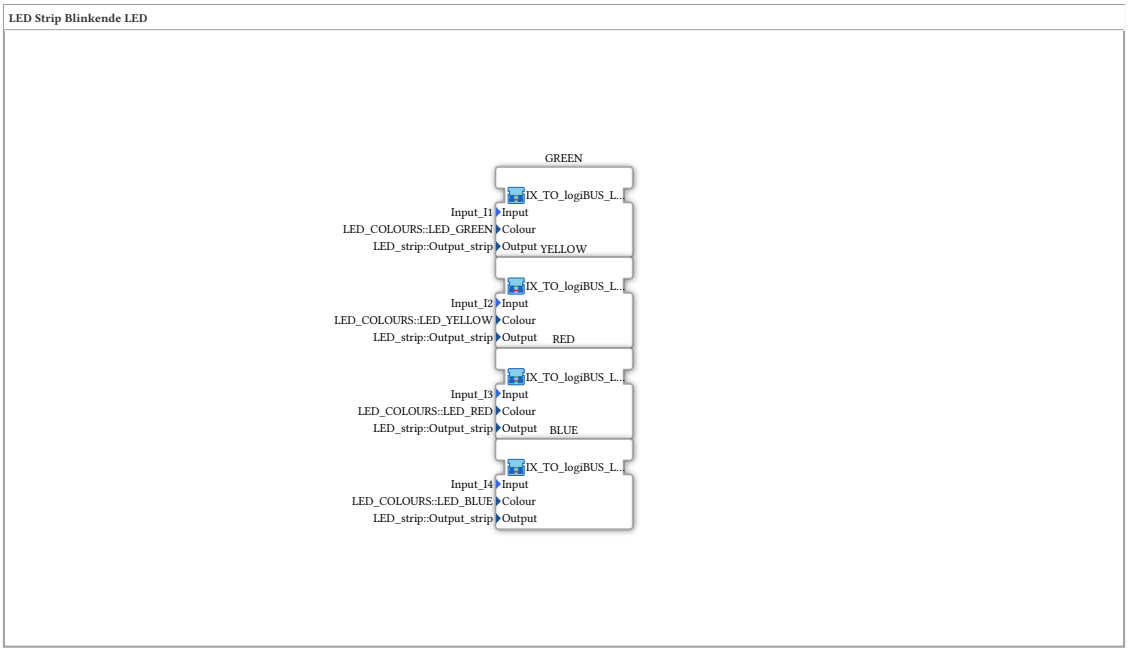
- Druck auf **Grün** ➡️ Streifen blitzt schnell grün.
- Druck auf **Rot** ➡️ Streifen wechselt sofort auf schnelles rotes Blitzen.

Dies ermöglicht eine sehr schnelle Programmierung von farbigen Status-Signalen.

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

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de
- 🌐 ESP32 & ESP32-S3 DevKit auf ms-muc-docs.de

2.2.1.219 Uebung_033: LED Strip Blinkende LED





Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_033`.

Übersicht

Dies ist die modulare Variante der Übung 032. Die Logik für einen farbigen LED-Kanal wurde in eine typisierte Sub-Applikation `Uebung_033_sub` ausgelagert. Das Hauptprogramm instanziiert diesen Typ viermal und weist ihm die Taster `I1` bis `I4` sowie die Zielfarben zu. Dies zeigt erneut die Vorteile der Wiederverwendbarkeit bei komplexen Hardware-Komponenten wie LED-Controllern.

2.2.1.220 Uebung_033_sub

Hinweis

Diese Übung existiert nicht mehr als eigene Übung. Der Baustein wurde in die Bibliothek verschoben und heißt jetzt IX_TO_logiBUS_LED_strip_QX. Beschreibung, Schnittstelle und Netzwerk stehen dort.

2.2.1.221 Uebung_034: Analog-Eingang auf PWM Ausgang



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_034`. Hier wird ein analoger Messwert genutzt, um die Leistung eines Aktors stufenlos zu regeln.

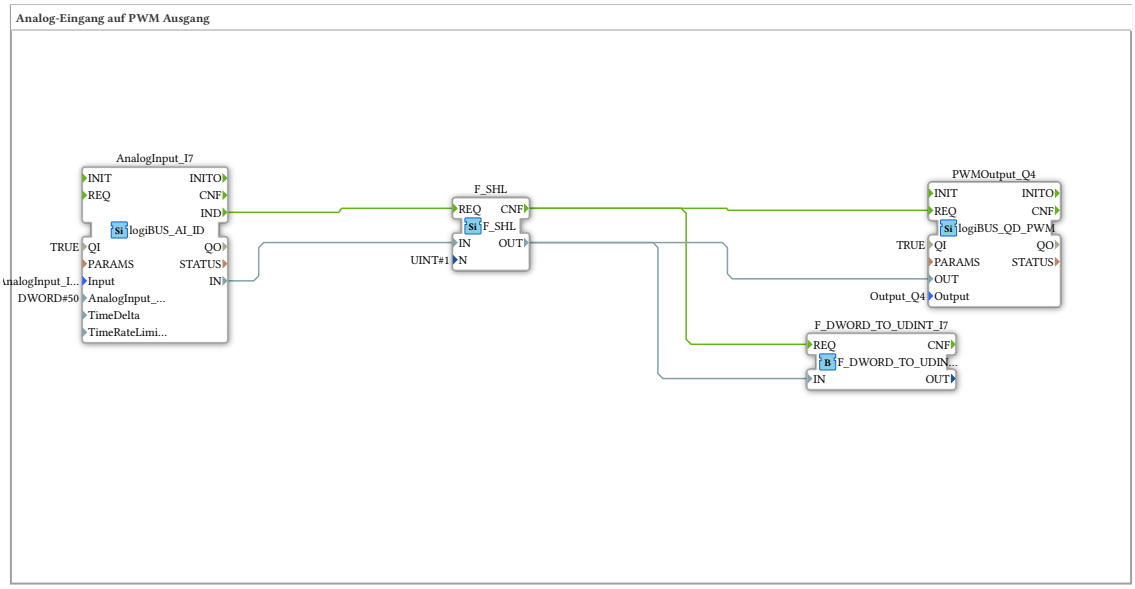
Ziel der Übung

Verbindung eines Analog-Eingangs (`logiBUS_AI`) mit einem PWM-Ausgang (`logiBUS_QD_PWM`). Es wird demonstriert, wie Datenwerte skaliert werden, um den Stellbereich eines Sensors auf den Leistungsbereich eines Aktors abzubilden.

Beschreibung und Komponenten

Die Subapplikation `Uebung_034.SUB` liest ein Potentiometer ein und steuert damit die Helligkeit einer Lampe oder die Drehzahl eines Motors.

Funktionsbausteine (FBs)



- **`AnalogInput_I7`**: Liest die Spannung am Eingang ein.
- **`F_SHL`**: Ein Schieberegister-Baustein (Shift Left). Er wird hier zur Skalierung genutzt, indem er den Eingangswert um ein Bit nach links verschiebt entspricht einer Multiplikation mit 2.
- **`PWMOutput_Q4`**: Ein pulsweitenmodulierter Ausgang zur Leistungsstellung.

Funktionsweise

1. Jede Änderung am analogen Eingang `I7` löst ein `IND`-Event aus.
2. Der Wert wird im `F_SHL` angepasst, um den gewünschten Zielbereich zu erreichen.
3. Das Ergebnis wird an den `OUT`-Port des PWM-Bausteins gesendet und über `REQ` aktiviert.
4. Der Aktor an `Q4` reagiert sofort auf die neue Vorgabe.

Anwendungsbeispiel

Licht-Dimmer oder Lüfter-Steuerung: Durch Drehen an einem physischen Potentiometer (`I7`) kann der Bediener die Helligkeit der Kabinenbeleuchtung oder die Stärke eines Gebläses (`Q4`) stufenlos regeln. Die Software sorgt für die latenzfreie Übertragung der Steuerbefehle.

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

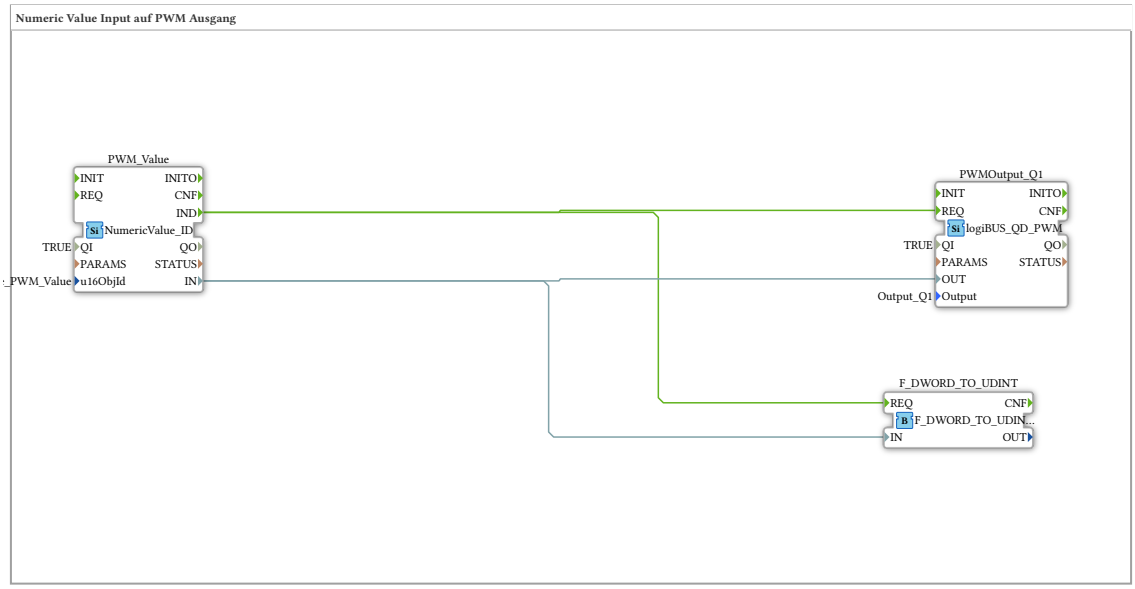
- 🌐 Das PWM-Signal & Infografik auf ms-muc-docs.de

2.2.1.222 Uebung_034a1_Q1: Numeric Value Input auf PWM Ausgang



Übersicht

In dieser Übung wird ein numerischer Wert direkt vom ISOBUS-Terminal eingelesen, um die Tastrate eines PWM-Ausgangs (`Q1`) zu steuern. Über das Objekt `InputNumber_PWM_Value` kann der Bediener am Bildschirm eine Zahl eingeben. Erst nach der Bestätigung mit „OK“ wird das Ereignis `IND` gefeuert und der neue Wert an die PWM-Hardware übertragen. Dies ermöglicht die präzise manuelle Vorgabe von Leistungen (z.B. Lüfterdrehzahl oder Lampenhelligkeit) direkt über das Display.



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

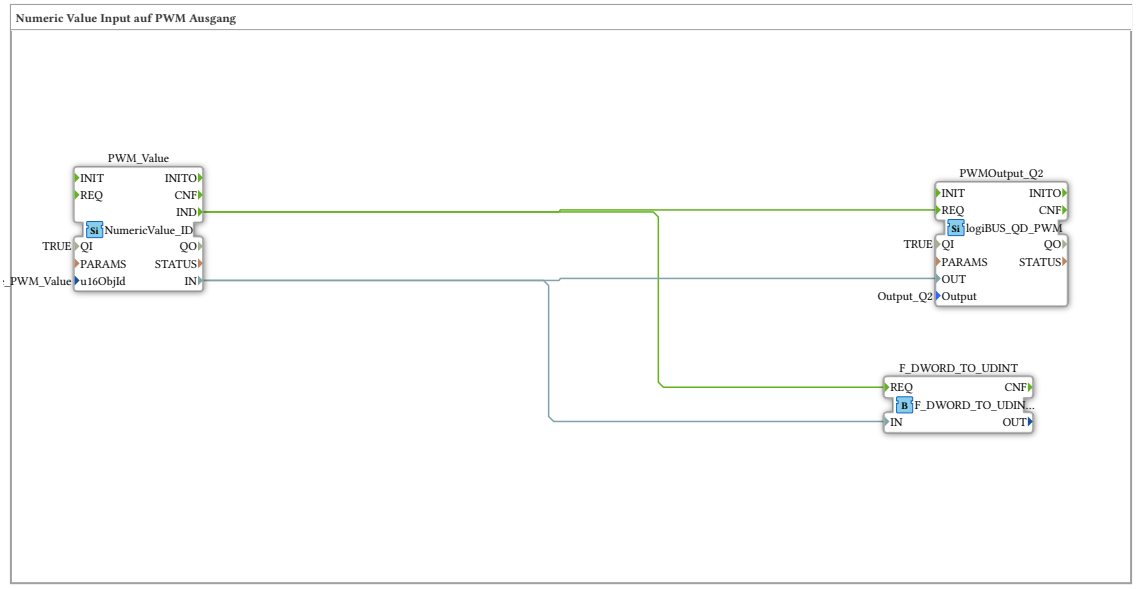
- 🌐 Das PWM-Signal & Infografik auf ms-muc-docs.de

2.2.1.223 Uebung_034a1_Q2: Numeric Value Input auf PWM Ausgang



Übersicht

Variante von Übung 034a1_Q1, konfiguriert für den Hardware-Ausgang `Q2`. Demonstriert die Skalierbarkeit der PWM-Ansteuerung über das Terminal.



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

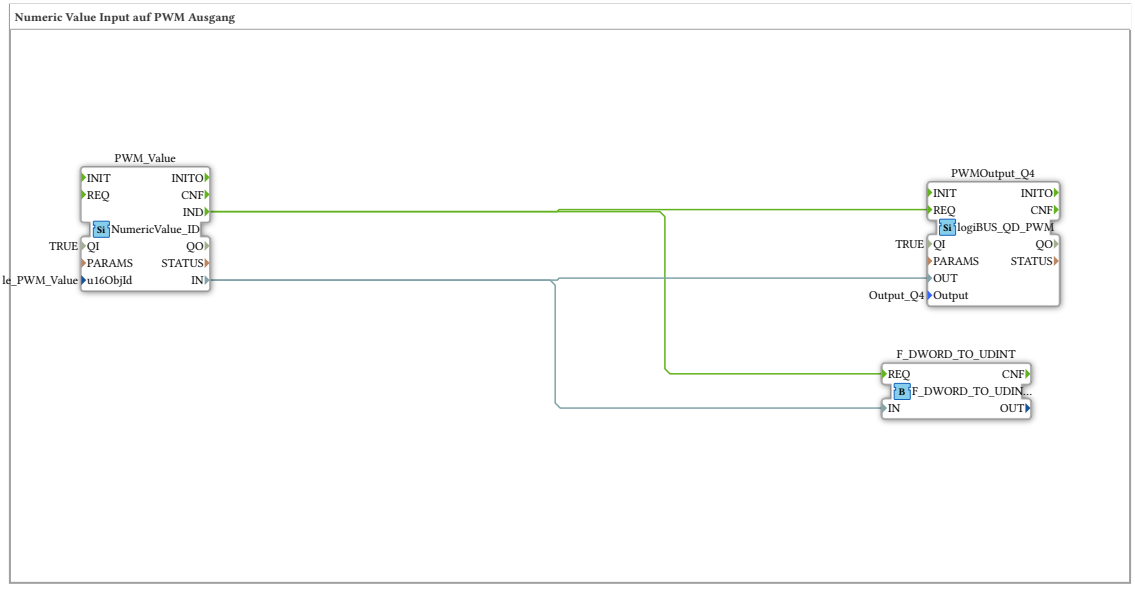
- 🌐 Das PWM-Signal & Infografik auf ms-muc-docs.de

2.2.1.224 Uebung_034a1_Q4: Numeric Value Input auf PWM Ausgang

NotebookLM

Übersicht

Variante von Übung 034a1_Q1, konfiguriert für den Hardware-Ausgang Q4.



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

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

2.2.1.225 Uebung_034b: LONG_PRESS_HOLD-Eingang auf PWM Ausgang mit Terminal-Ausgabe

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_034b`. Hier wird die PWM-Leistung über Taster-Interaktionen („Gas geben“) gesteuert.

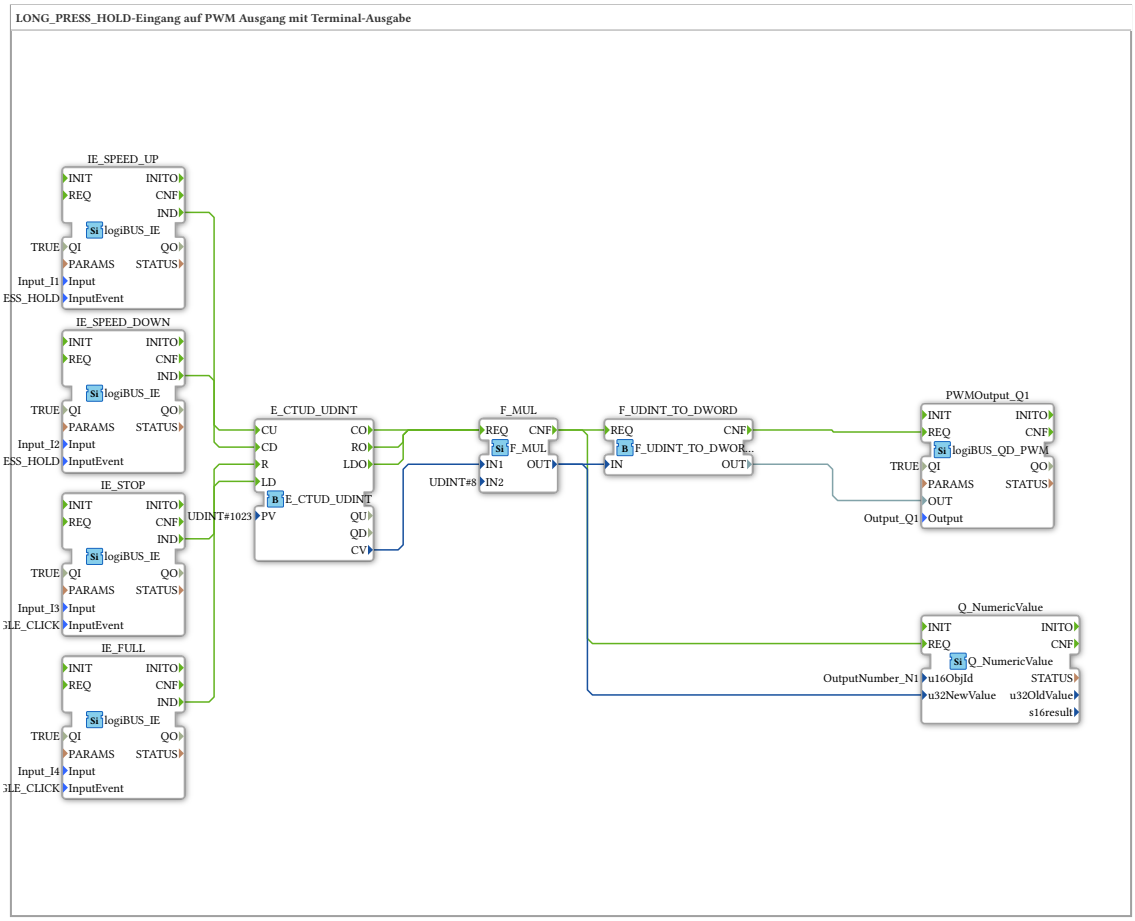
Ziel der Übung

Kombination von repetierenden Ereignissen (`HOLD`) und Zählern zur Steuerung einer PWM-Stufe. Der Nutzer kann die Leistung durch Festhalten eines Tasters stufenweise erhöhen oder verringern.

Beschreibung und Komponenten

In `Uebung_034b.SUB` wird ein Up/Down-Zähler als digitaler Integrator genutzt.

Funktionsbausteine (FBs)



- **`IE_SPEED_UP`**: Sendet alle 200ms ein Event, solange Taster **I1** gehalten wird.
- **`IE_SPEED_DOWN`**: Sendet alle 200ms ein Event, solange Taster **I2** gehalten wird.
- **`E_CTUD_UDINT`**: Speichert den aktuellen „Leistungs-Zählerstand“.
- **`F_MUL`**: Skaliert den Zählerstand (hier Faktor 8) auf den Zielbereich für den PWM-Baustein.
- **`PWMOutput_Q1`**: Der Leistungsausgang.

Funktionsweise

1. **Steigern**: Der Bediener hält **I1** gedrückt. Der Zähler zählt alle 200ms einen Schritt hoch. Die Lampe an `Q1` wird stufenweise heller.
2. **Senken**: Der Bediener hält **I2** gedrückt. Die Lampe wird stufenweise dunkler.

3. **Schnell-Wahl**: Taster **I3** (Stopp) setzt den Wert sofort auf
4. Taster **I4** (Full) lädt den Zähler sofort auf das Maximum.

Dies ermöglicht eine sehr feinfühlige Steuerung von Antrieben oder Beleuchtungen über einfache digitale Taster.

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

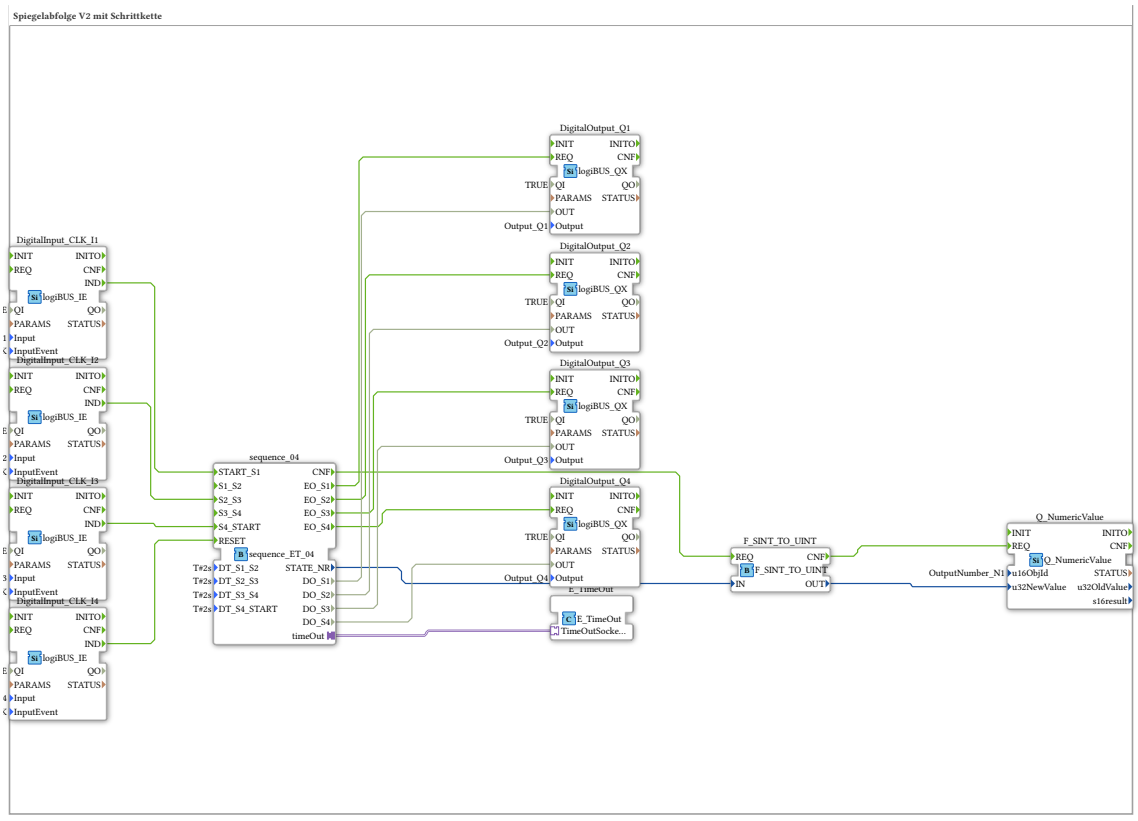
- 🌐 E_CTU Event Counter Baustein auf ms-muc-docs.de
- 🌐 Das PWM-Signal & Infografik auf ms-muc-docs.de

2.2.1.226 Uebung_035: Spiegelabfolge V2 mit Schrittkette



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_035`. Hier wird die Steuerung von komplexen Abläufen mittels eines Sequenzers (Schrittkette) demonstriert.

🎧 Podcast



- Automatisierung entschlüsselt: Leiten, Steuern, Regeln – Die unsichtbare Sprache der Technik (DIN IEC 60050-351)
- Infineon CAN-Transceiver TLE9250V versus TLE9351VSJ
- Infineon TLE9351VSJ der unsichtbare Auto-Bodyguard
- JBCs Löt-Geheimnis: 350 Grad in 2 Sekunden und warum die Spitze über Effizienz und Lebensdauer entscheidet

---

Ziel der Übung

Verwendung des Bausteins `sequence_ET_04`. Es wird gezeigt, wie ein Prozess in vier Phasen (`S1` bis `S4`) unterteilt wird, wobei die Übergänge sowohl durch Ereignisse (Events) als auch durch Zeit (Timer) ausgelöst werden können.

Beschreibung und Komponenten

Die Subapplikation `Uebung_035.SUB` steuert 4 Ausgänge in einer festen Reihenfolge.

Funktionsbausteine (FBs)

- **`sequence_04`**: Der Sequenzer-Baustein. Er verwaltet die Logik der Schritte.
- **Parameter `DT_S1_S2` etc.**: Definieren die maximale Verweildauer in einem Schritt (hier jeweils 2 Sekunden).
- **`Q_NumericValue`**: Zeigt den aktuellen Schritt (1-4) auf dem Terminal an.
- **`E_TimeOut`**: Überwacht den Ablauf.

Funktionsweise

1. **Start**: Taster **I1** triggert `START_S1`. Lampe `Q1` geht an.
2. **Übergang**: Nach 2 Sekunden (oder durch ein Event am entsprechenden Port) springt der Sequenzer zu Schritt 2. `Q1` geht aus, `Q2` geht an.
3. **Fortsetzung**: Der Prozess läuft bis Schritt 4 durch.
4. **Reset**: Taster **I4** kann den Ablauf jederzeit abbrechen und alle Ausgänge deaktivieren.

Anwendungsbeispiel

Automatischer Reinigungszyklus: Ein Knopfdruck startet das Programm:

1. Ventile spülen (2s),
2. Reinigungsmittel einlassen (2s),
3. Einwirken (2s),
4. Klarspülen (2s). Die Schrittkette garantiert, dass die Phasen exakt nacheinander und niemals gleichzeitig ablaufen.

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

- 🌐 Interaktiver JBC-Lötspitzen Guide & Infografik auf ms-muc-docs.de

2.2.1.227 Uebung_035a: Ampelschaltung International



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_035a`. Hier wird die Steuerung einer Lichtsignalanlage (Ampel) mittels einer zeitgesteuerten Schrittkette realisiert.

Ziel der Übung

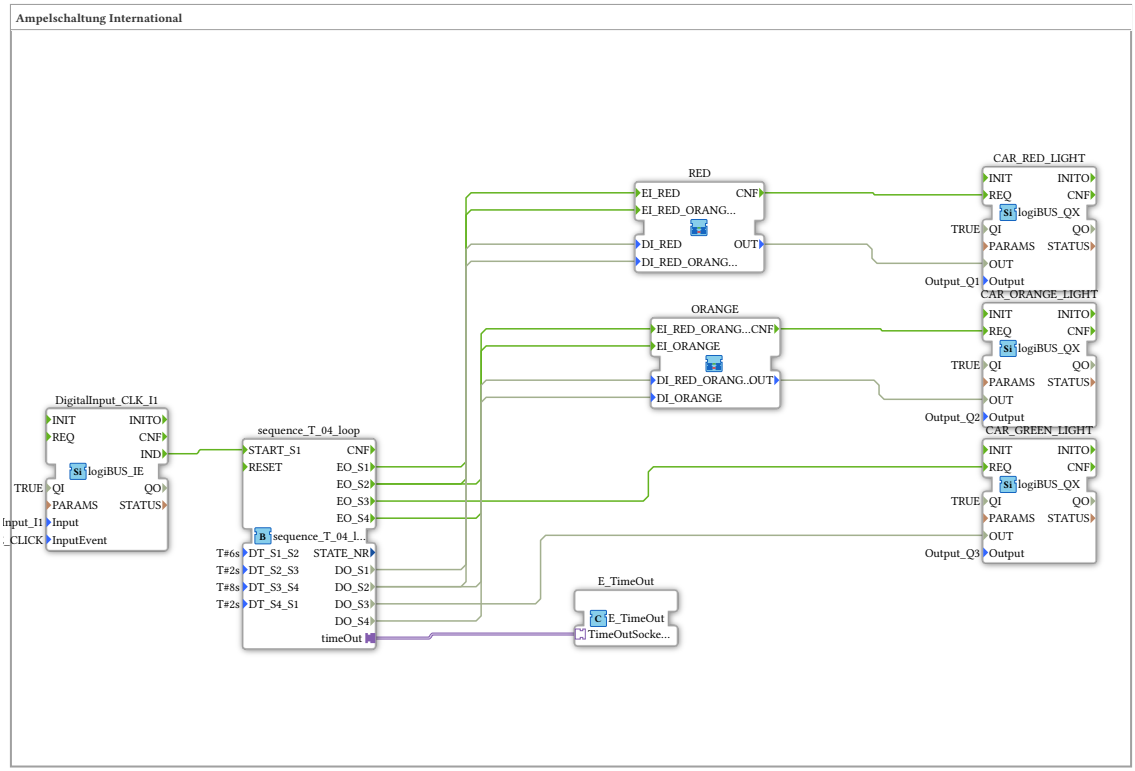
Realisierung eines komplexen Zeitablaufs mit überlappenden Zuständen. Es wird die Standard-Abfolge für Deutschland simuliert: Rot ➡️ Rot-Gelb ➡️ Grün ➡️ Gelb ➡️ Rot.

Beschreibung und Komponenten

In `Uebung_035a.SUB` wird ein 4-Schritt-Sequenzer als Taktgeber genutzt.

Funktionsweise

Die Herausforderung liegt in den Misch-Zuständen (z.B. Rot und Gelb leuchten gleichzeitig). Dies wird durch logische ODER-Gatter in Sub-Applikationen (`RED`, `ORANGE`) gelöst:



1. **Schritt 1 (Rot)**: Nur der Rot-Ausgang ist aktiv (6s).
2. **Schritt 2 (Rot-Gelb)**: Das Event triggert beide Lampen (2s).
3. **Schritt 3 (Grün)**: Nur Grün leuchtet (8s).
4. **Schritt 4 (Gelb)**: Nur Gelb leuchtet (2s).

Danach beginnt der Zyklus von vorn. Dies demonstriert die Kombination von sequenziellem Ablauf und kombinatorischer Logik.

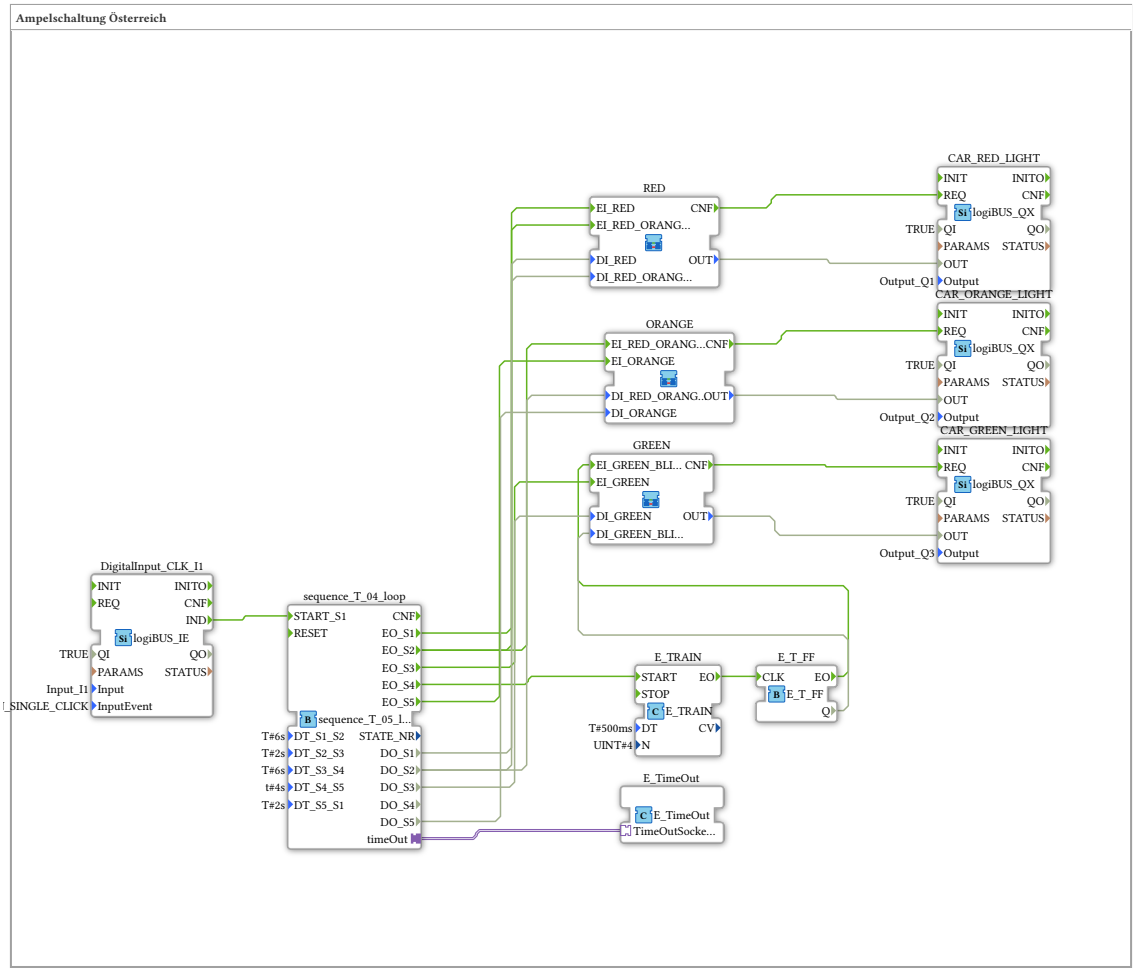
2.2.1.228 Uebung_035a2: Ampelschaltung Österreich

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_035a2`. Hier wird die Ampelsteuerung um die in einigen Ländern (z.B. Österreich) übliche Grün-Blinkphase erweitert.

Übersicht

Unter Verwendung eines 5-Schritt-Sequenzers wird ein zusätzlicher Zustand „Grün Blinken“ eingefügt. Nach der Grünphase (Schritt 3) startet der Sequenzer einen `E_TRAIN` Baustein (Schritt 4). Dieser erzeugt 4 kurze Impulse im 500ms Takt, die über ein Toggle-Flip-Flop die grüne Lampe blinken lassen. Erst nach Abschluss dieser Blink-Sequenz schaltet das System in die Gelbphase (Schritt 5). Dies zeigt die nahtlose Integration von Sub-Sequenzen innerhalb einer übergeordneten Schrittkette.

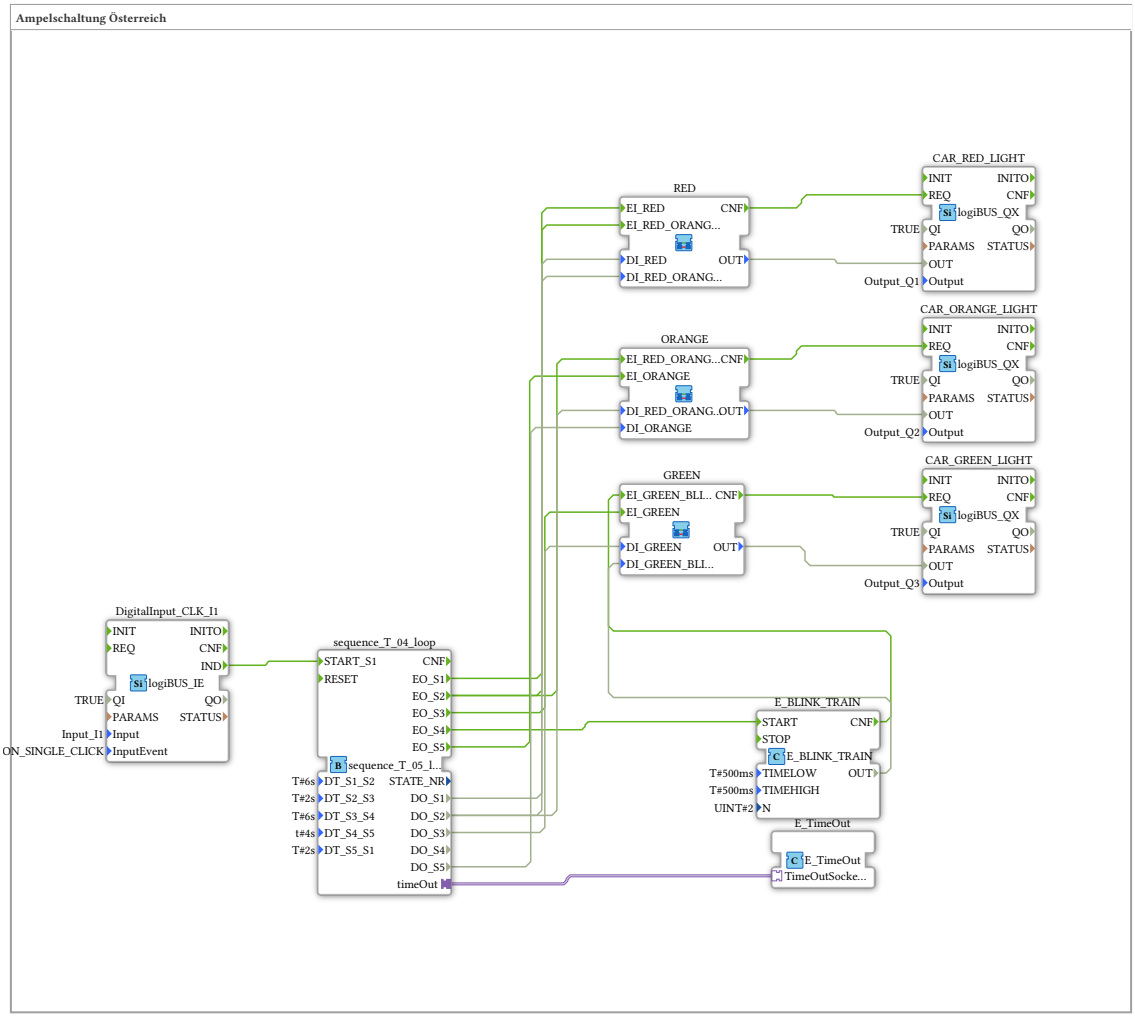


2.2.1.229 Uebung_035a3: Ampelschaltung Österreich

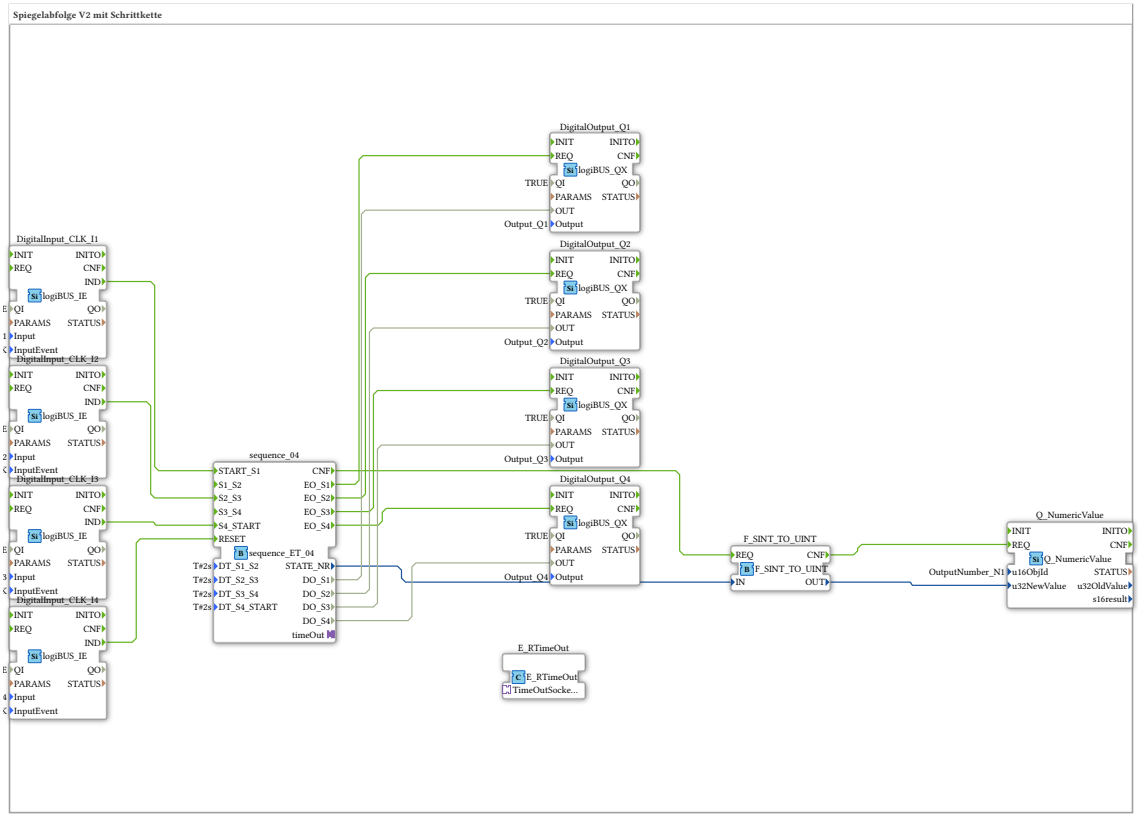


Übersicht

Strukturelle Variante der Übung 035a2. Anstelle des `E_TRAIN` Bausteins wird hier der spezialisierte `E_BLINK_TRAIN` genutzt, um die Grün-Blinkphase noch präziser zu steuern. Die Logik der Zustandsüberlappung (Rot-Gelb) wird weiterhin über Sub-Applikations-ODER-Gatter realisiert.


Ampelschaltung Österreich
RED
EI_RED
EI_RED_ORANG...
DI_RED
DI_RED_ORANG...
CNF
OUT
ORANGE
EI_RED_ORANG...
EI_ORANGE
DI_RED_ORANG...
DI_ORANGE
CNF
OUT
GREEN
EI_GREEN_BLI...
EI_GREEN
DI_GREEN
DI_GREEN_BLI...
CNF
OUT
CAR_RED_LIGHT
CAR_ORANGE_LIGHT
CAR_GREEN_LIGHT
INIT
REQ
logiBUS_QX
QI
PARAMS
OUT
INITO
CNF
QO
STATUS
Output
TRUE
Output_Q1
Output_Q2
Output_Q3
DigitalInput_CLK_I1
INIT
REQ
logiBUS_IE
QI
PARAMS
Input
InputEvent
INITO
CNF
IND
QO
STATUS
TRUE
Input_I1
ON_SINGLE_CLICK
sequence_T_04_loop
START_S1
RESET
EO_S1
EO_S2
EO_S3
EO_S4
EO_S5
sequence_T_05_l...
DT_S1_S2
DT_S2_S3
DT_S3_S4
DT_S4_S5
DT_S5_S1
STATE_NR
DO_S1
DO_S2
DO_S3
DO_S4
DO_S5
timeOut
T#6s
T#2s
T#6s
t#4s
T#2s
E_BLINK_TRAIN
START
STOP
E_BLINK_TRAIN
TIMELOW
TIMEHIGH
N
CNF
OUT
T#500ms
T#500ms
UINT#2
E_TimeOut
E_TimeOut
TimeOutSocke...

2.2.1.230 Uebung_035b: Spiegelabfolge V2 mit Schrittkette



• – ○ • – ○ • – ○ •

Einleitung

In dieser Übung wird eine Schrittkettensteuerung (Sequencer) realisiert, die als „Spiegelabfolge V2“ bezeichnet wird. Das Ziel ist die sequentielle Ansteuerung von vier digitalen Ausgängen (Q1 bis Q4). Zusätzlich wird der aktuelle Status der Schrittkette als numerischer Wert auf einer Benutzeroberfläche visualisiert. Die Steuerung erfolgt über digitale Eingänge, welche die Sequenz starten, beeinflussen oder zurücksetzen können.

Verwendete Funktionsbausteine (FBs)

In dieser Anwendung werden verschiedene Funktionsbausteine aus den Bibliotheken `logiBUS`, `isobus` und `iec61131` verwendet.

Haupt-Bausteine

- **DigitalInput_CLK_I1** bis **DigitalInput_CLK_I4** (`logiBUS::io::DI::logiBUS_IE`): Diese Bausteine verarbeiten die physischen Eingabesignale (Taster). Sie sind so konfiguriert, dass sie auf das Ereignis `BUTTON_SINGLE_CLICK` reagieren.
- **DigitalOutput_Q1** bis **DigitalOutput_Q4** (`logiBUS::io::DQ::logiBUS_QX`): Diese Bausteine steuern die physischen Ausgänge (Lampen/Aktoren).
- **Q_NumericValue** (`isobus::UT::Q::Q_NumericValue`): Dient zur Anzeige eines numerischen Wertes auf einem Universal Terminal (UT). Hier wird die Objekt-ID `OutputNumber_N1` verwendet.
- **F_SINT_TO_UINT** (`iec61131::conversion::F_SINT_TO_UINT`): Ein Konvertierungsbaustein, der den Datentyp Short Integer (SINT) in Unsigned Integer (UINT) umwandelt, um ihn kompatibel für die Anzeige zu machen.
- **E_RTimeOut** (`iec61499::events::E_RTimeOut`): *Hinweis: Dieser Baustein ist im Netzwerk platziert, aber im aktuellen Stand der Übung nicht verdrahtet (laut Kommentar ein „TODO“ für ein zukünftiges Beispiel).*

Sub-Bausteine: sequence_ET_04

Dies ist der zentrale Logikbaustein der Übung.

- **Typ**: `logiBUS::utils::sequence::combi::sequence_ET_04`
- **Beschreibung**: Eine Schrittkette mit 4 Schritten, die Zeit- und Ereignisgesteuert arbeitet.
- **Parameter**:
  - `DT_S1_S2` = `T#2s`: Zeitdauer/Verzögerung zwischen Schritt 1 und 2.
  - `DT_S2_S3` = `T#2s`: Zeitdauer/Verzögerung zwischen Schritt 2 und 3.
  - `DT_S3_S4` = `T#2s`: Zeitdauer/Verzögerung zwischen Schritt 3 und 4.
  - `DT_S4_START` = `T#2s`: Zeitdauer/Verzögerung nach Schritt 4 bis zum Neustart.
- **Ereigniseingänge (Verwendet)**:
  - `START_S1`: Startet die Sequenz bei Schritt 1.
  - `S2_S3`: Trigger für den Übergang oder Einfluss auf den Schrittwechsel 2 zu 3.
  - `S4_START`: Trigger für den Übergang oder Einfluss auf den Schrittwechsel 4 zurück zum Start.
  - `RESET`: Setzt die Schrittkette zurück.
- **Ausgänge**:
  - `EO_S1` bis `EO_S4`: Ereignisausgänge, die feuern, wenn der jeweilige Schritt aktiv wird.
  - `DO_S1` bis `DO_S4`: Datenausgänge (BOOL), die den Status des jeweiligen Schritts führen.
  - `STATE_NR`: Gibt die aktuelle Schrittnummer als Zahl aus.

Programmablauf und Verbindungen

Der Ablauf der Übung wird durch die Interaktion der Taster mit dem Sequencer-Baustein bestimmt:

1. **Steuerung der Sequenz**:
   - Der Taster **I1** ist mit dem Eingang `START_S1` verbunden. Ein Klick startet die Schrittkette.
   - Der Taster **I2** greift in den Übergang `S2_S3` ein.
   - Der Taster **I3** greift in den Übergang `S4_START` ein.
   - Der Taster **I4** ist mit dem `RESET`-Eingang verbunden und stoppt/resettet die gesamte Sequenz.
2. **Ausgabe der Signale**:
   - Der Sequencer `sequence_04` schaltet je nach aktivem Schritt die Ausgänge:
     - Schritt 1 aktiviert **Q1**.
     - Schritt 2 aktiviert **Q2**.
     - Schritt 3 aktiviert **Q3**.
     - Schritt 4 aktiviert **Q4**.
   - Die Zeitparameter (T#2s) im Sequencer deuten darauf hin, dass die Schritte eine definierte Laufzeit haben oder automatische Übergänge nach 2 Sekunden stattfinden, sofern sie nicht durch die Eingänge übersteuert werden.
3. **Visualisierung**:
   - Die aktuelle Schrittnummer (`STATE_NR` vom Sequencer) wird an den Konverter `F_SINT_TO_UINT` gesendet.
   - Der konvertierte Wert (`u32NewValue`) wird an den Baustein `Q_NumericValue` übergeben, um die Nummer des aktiven Schritts auf dem Display (Objekt `OutputNumber_N1`) anzuzeigen.

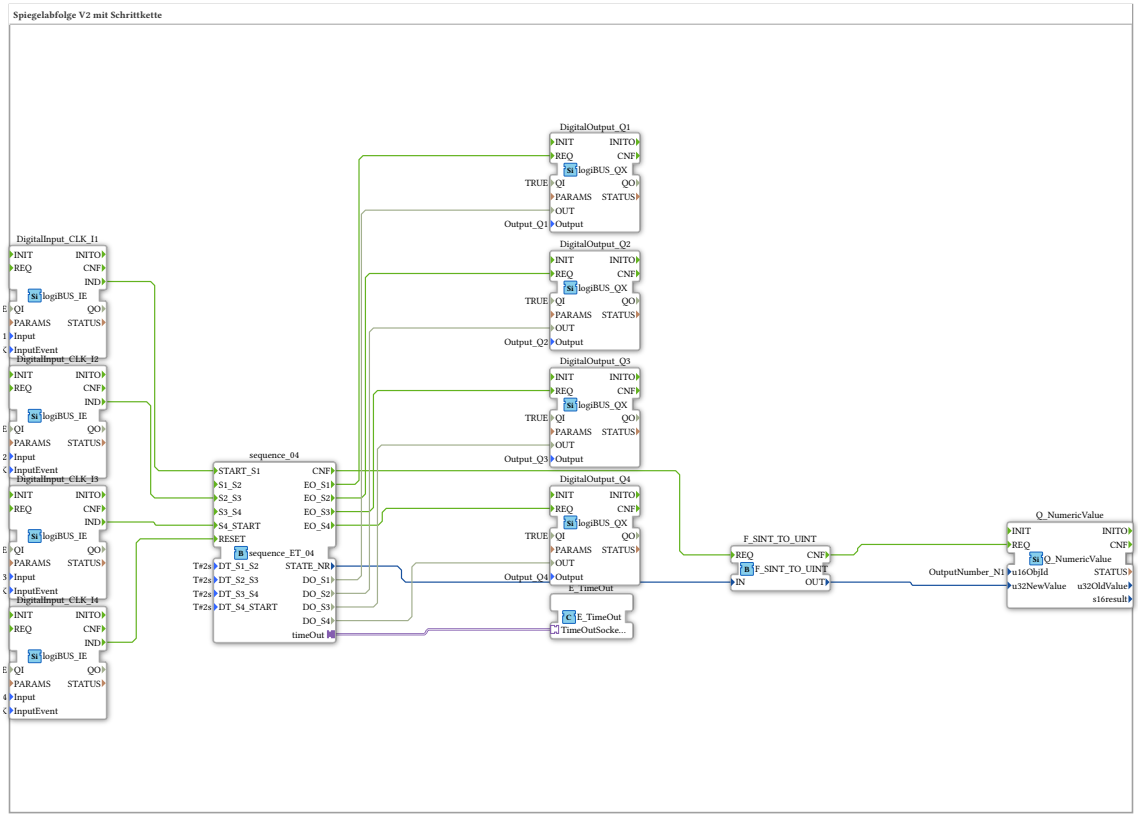
Zusammenfassung

Die Übung **Uebung_035b** demonstriert die Implementierung einer komplexeren Schrittkettensteuerung (`sequence_ET_04`) innerhalb der 4diac-IDE. Sie verknüpft manuelle Benutzereingaben (Start, Reset, spezifische Übergangstrigger) mit zeitbasierten Parametern, um vier Ausgänge sequenziell zu schalten. Gleichzeitig wird der interne Zustand der Logik (die Schrittnummer) für den Benutzer auf einem Display visualisiert.

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

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

2.2.1.231 Uebung_035c: Spiegelabfolge V2 mit Schrittkette



(Hier Bild der Übung einfügen, falls vorhanden)

• – ○ • – ○ • – ○ •

Einleitung

Die Übung **Uebung_035c** („Spiegelabfolge V2 mit Schrittkette“) demonstriert die Steuerung einer sequenziellen Abfolge (Schrittkette) mit vier Zuständen. Dabei kommen sowohl zeitgesteuerte als auch ereignisgesteuerte Übergänge zum Einsatz. Der aktuelle Status der Schrittkette wird über digitale Ausgänge (LEDs) visualisiert und die Nummer des aktiven Schritts wird auf einer numerischen Anzeige ausgegeben.

Verwendete Funktionsbausteine (FBs)

In dieser Übung werden verschiedene Standard- und Logikbausteine verwendet, um die Ein- und Ausgabe sowie die Ablaufsteuerung zu realisieren.

Haupt-Bausteine

- **DigitalInput_CLK_I1 bis I4** (`logiBUS::io::DI::logiBUS_IE`):
  - Dienen zum Einlesen der Taster-Signale (Input_I1 bis Input_I4).
  - Konfiguriert auf das Ereignis `BUTTON_SINGLE_CLICK`.
- **DigitalOutput_Q1 bis Q4** (`logiBUS::io::DQ::logiBUS_QX`):
  - Steuern die physikalischen Ausgänge (Output_Q1 bis Output_Q4) an, um den aktiven Schritt anzuzeigen.
- **Q_NumericValue** (`isobus::UT::Q::Q_NumericValue`):
  - Dient zur Visualisierung eines numerischen Wertes auf dem Display (Objekt-ID `OutputNumber_N1`).
- **F_SINT_TO_UINT** (`iec61131::conversion::F_SINT_TO_UINT`):
  - Konvertiert die Schrittnummer (SINT) in einen vorzeichenlosen Integer (UINT), damit dieser vom `Q_NumericValue` Baustein verarbeitet werden kann.
- **E_TimeOut** (`iec61499::events::E_TimeOut`):
  - Ein Systembaustein für Zeitfunktionen, der über einen Adapter mit der Schrittkette verbunden ist.

Sub-Bausteine: sequence_04

Dieser Baustein ist das Herzstück der logischen Steuerung.

- **Typ**: `logiBUS::utils::sequence::combi::sequence_ET_04`
- **Verwendete interne FBs**: (Logik kapselt eine Zustandsmaschine mit Zeitgliedern)
  - **Parameter**:
    - `DT_S1_S2` = `T#2s`: Verzögerungszeit für den Übergang von Schritt 1 zu 2.
    - `DT_S2_S3` = `T#2s`: Verzögerungszeit für den Übergang von Schritt 2 zu 3.
    - `DT_S3_S4` = `T#2s`: Verzögerungszeit für den Übergang von Schritt 3 zu 4.
    - `DT_S4_START` = `T#2s`: Verzögerungszeit für den Rücksprung zum Start.
  - **Ereigniseingänge**:
    - `START_S1`: Startet die Sequenz bei Schritt 1 (verbunden mit Taster I1).
    - `S2_S3`: Trigger für den Übergang von Schritt 2 zu 3 (verbunden mit Taster I2).
    - `S4_START`: Trigger für den Neustart nach Schritt 4 (verbunden mit Taster I3).
    - `RESET`: Setzt die Kette zurück (verbunden mit Taster I4).
  - **Datenausgänge**:
    - `DO_S1` bis `DO_S4`: Statussignale für die Ausgänge Q1 bis Q4.
    - `STATE_NR`: Gibt die aktuelle Schrittnummer als Zahl aus.

Programmablauf und Verbindungen

Die Logik kombiniert automatische Zeitübergänge mit manuellen Benutzereingriffen.

1. **Starten der Sequenz**:
   - Durch Betätigen von **Taster I1** wird das Ereignis `START_S1` ausgelöst.
   - Die Schrittkette springt in **Zustand 1**.
   - Der Ausgang **Q1** wird aktiviert.
2. **Automatische und Manuelle Übergänge**:
   - **S1 -> S2**: Da kein explizites Ereignis für diesen Übergang verdrahtet ist, erfolgt der Wechsel zu **Zustand 2** (Q2 an) automatisch nach Ablauf der Zeit `DT_S1_S2` (2 Sekunden).
   - **S2 -> S3**: Der Übergang zu **Zustand 3** (Q3 an) erfordert eine manuelle Bestätigung durch **Taster I2**, da dieser am Eingang `S2_S3` angeschlossen ist. Die Zeit `DT_S2_S3` dient hierbei vermutlich als Mindestwartezeit oder Timeout-Basis.
   - **S3 -> S4**: Der Wechsel zu **Zustand 4** (Q4 an) erfolgt wieder automatisch nach Ablauf von `DT_S3_S4` (2 Sekunden), da kein Taster-Event angeschlossen ist.
   - **S4 -> Start**: Um die Kette von Zustand 4 wieder neu zu starten, muss **Taster I3** betätigt werden (Eingang `S4_START`).
3. **Visualisierung**:
   - Parallel zu den LEDs wird die aktuelle Schrittnummer (`STATE_NR`) über den Konverter an den Baustein `Q_NumericValue` gesendet und auf dem Display (OutputNumber_N1) angezeigt.
4. **Reset**:
   - Der **Taster I4** ist mit dem `RESET`-Eingang verbunden und setzt die gesamte Schrittkette jederzeit in den Grundzustand zurück (alle Ausgänge aus).

Zusammenfassung

Diese Übung vertieft das Verständnis für komplexe Schrittketten in IEC 61499. Sie zeigt auf, wie manuelle Eingriffe (Trigger durch Taster) mit automatischen Zeitabläufen gemischt werden können. Zudem wird die Verarbeitung von numerischen Daten zur Zustandsanzeige und die Konvertierung von Datentypen (`SINT_TO_UINT`) praktisch angewendet.

2.2.1.232 Uebung_036: Spiegelabfolge V2 mit Schrittkette



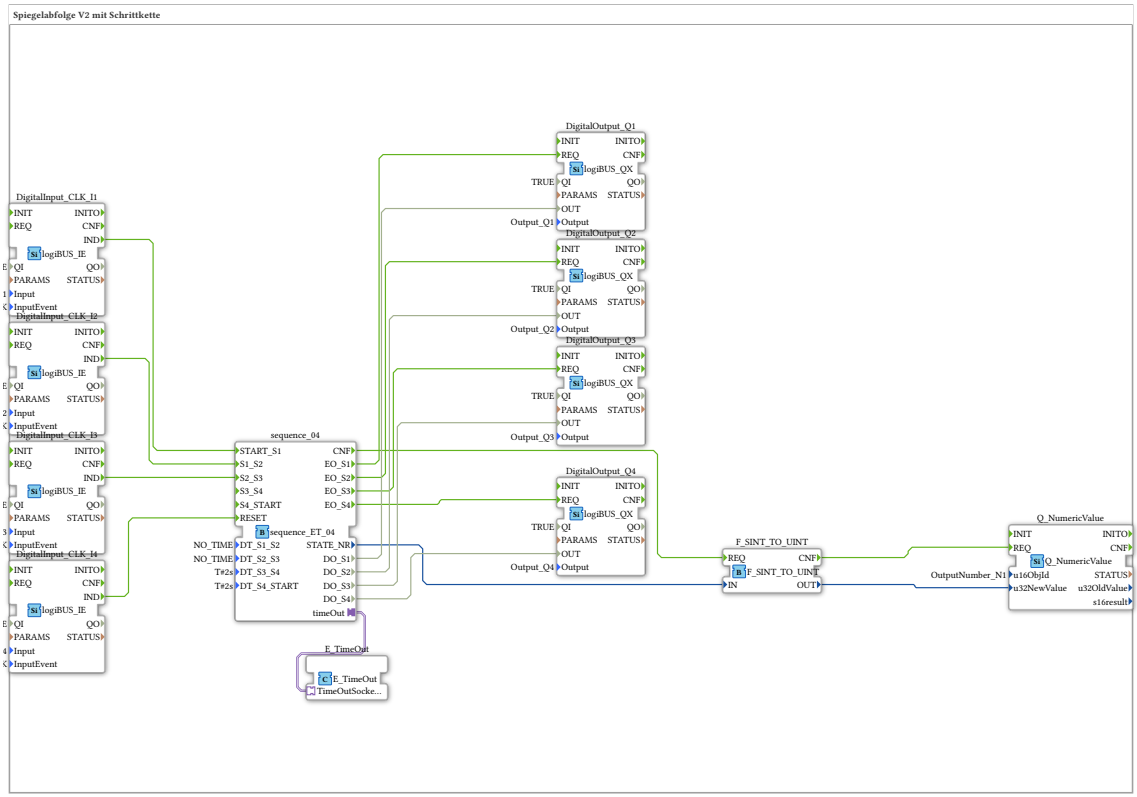
Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_036`. Im Gegensatz zu Übung 035 liegt hier der Fokus auf der manuellen Weiterschaltung durch Ereignisse.

Ziel der Übung

Realisierung einer Schrittkette ohne automatische Zeitübergänge.

Funktionsweise

In `Uebung_036.SUB` sind die Zeitparameter `DT_S1_S2` und `DT_S2_S3` auf den Wert `NO_TIME` gesetzt.



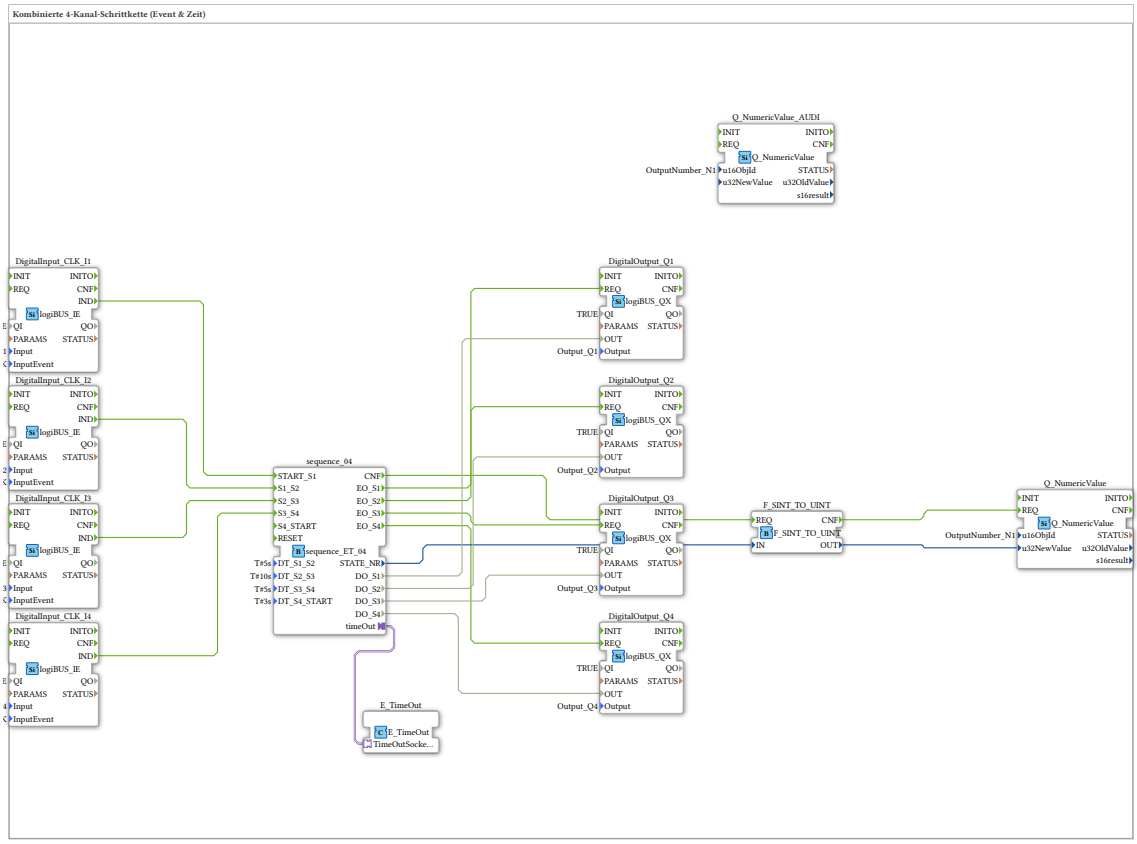
Das bedeutet: Der Sequenzer bleibt in Schritt 1 stehen, bis er ein explizites Ereignis am Eingang `S1_S2` erhält. In dieser Übung wird dies durch den Taster **I2** ausgelöst. Analog schaltet Taster **I3** von Schritt 2 zu 3 weiter. Erst die letzten Schritte nutzen wieder die Zeitautomatik (2s).

Anwendungsbeispiel

Manueller Bestückungsprozess: Ein Mitarbeiter legt ein Teil ein und drückt „Fertig“ (`I2`). Die Maschine führt den ersten Bearbeitungsschritt aus. Danach wartet sie wieder auf die Freigabe des Mitarbeiters (`I3`), bevor sie weitermacht. Die Schrittkette passt sich so dem Arbeitstempo des Menschen an.

2.2.1.233 Uebung_036b: Kombinierte 4-Kanal-Schrittkette (Event & Zeit)

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_036b (Kombinierte 4-Kanal-Schrittkette (Event & Zeit)).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Kombinierte 4-Kanal-Schrittkette (Event & Zeit)**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_036b.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QX.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
- **DigitalOutput_Q2**: Instanz des Typs logiBUS::io::DQ::logiBUS_QX.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q2
- **DigitalOutput_Q3**: Instanz des Typs logiBUS::io::DQ::logiBUS_QX.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q3
- **DigitalOutput_Q4**: Instanz des Typs logiBUS::io::DQ::logiBUS_QX.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q4
- **DigitalInput_CLK_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **DigitalInput_CLK_I2**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I2
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **DigitalInput_CLK_I3**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.

  - Parameter QI = TRUE
  - Parameter Input = Input_I3
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **DigitalInput_CLK_I4**: Instanz des Typs logiBUS::io::DI::logiBUS_IE.
  - Parameter QI = TRUE
  - Parameter Input = Input_I4
  - Parameter InputEvent = BUTTON_SINGLE_CLICK
- **Q_NumericValue_AUDI**: Instanz des Typs isobus::UT::Q::Q_NumericValue.
  - Parameter u16ObjId = OutputNumber_N1
- **E_TimeOut**: Instanz des Typs iec61499::events::E_TimeOut.
- **sequence_04**: Instanz des Typs logiBUS::utils::sequence::combi::sequence_ET_04.
  - Parameter DT_S1_S2 = T#5s
  - Parameter DT_S2_S3 = T#10s
  - Parameter DT_S3_S4 = T#5s
  - Parameter DT_S4_START = T#3s
- **Q_NumericValue**: Instanz des Typs isobus::UT::Q::Q_NumericValue.
  - Parameter u16ObjId = OutputNumber_N1
- **F_SINT_TO_UINT**: Instanz des Typs iec61131::conversion::F_SINT_TO_UINT.

Verbindungen und Schnittstellen

Adapterverbindungen:

- sequence_04.timeOut -> E_TimeOut.TimeOutSocket

Ereignisverbindungen:

- DigitalInput_CLK_I1.IND -> sequence_04.START_S1
- DigitalInput_CLK_I2.IND -> sequence_04.S1_S2
- DigitalInput_CLK_I3.IND -> sequence_04.S2_S3
- DigitalInput_CLK_I4.IND -> sequence_04.S3_S4
- F_SINT_TO_UINT.CNF -> Q_NumericValue.REQ
- sequence_04.CNF -> F_SINT_TO_UINT.REQ
- sequence_04.EO_S1 -> DigitalOutput_Q1.REQ
- sequence_04.EO_S2 -> DigitalOutput_Q2.REQ
- sequence_04.EO_S3 -> DigitalOutput_Q3.REQ
- sequence_04.EO_S4 -> DigitalOutput_Q4.REQ

Datenverbindungen:

- sequence_04.STATE_NR -> F_SINT_TO_UINT.IN
- sequence_04.DO_S1 -> DigitalOutput_Q1.OUT
- sequence_04.DO_S2 -> DigitalOutput_Q2.OUT
- sequence_04.DO_S3 -> DigitalOutput_Q3.OUT
- sequence_04.DO_S4 -> DigitalOutput_Q4.OUT
- F_SINT_TO_UINT.OUT -> Q_NumericValue.u32NewValue

Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

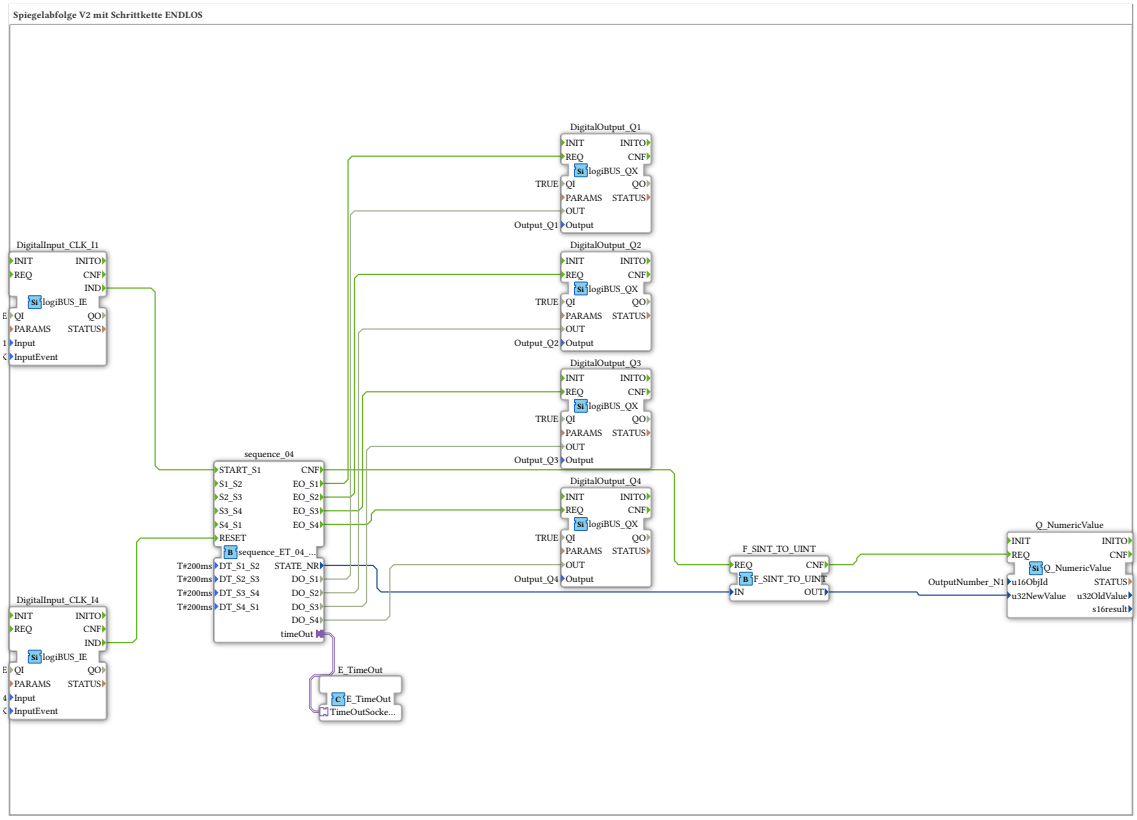
Zusammenfassung

Die Übung Uebung_036b demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.2.1.234 Uebung_037: Spiegelabfolge V2 mit Schrittkette ENDLOS

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_037`. Hier wird eine zyklisch wiederkehrende Abfolge programmiert.

Übersicht



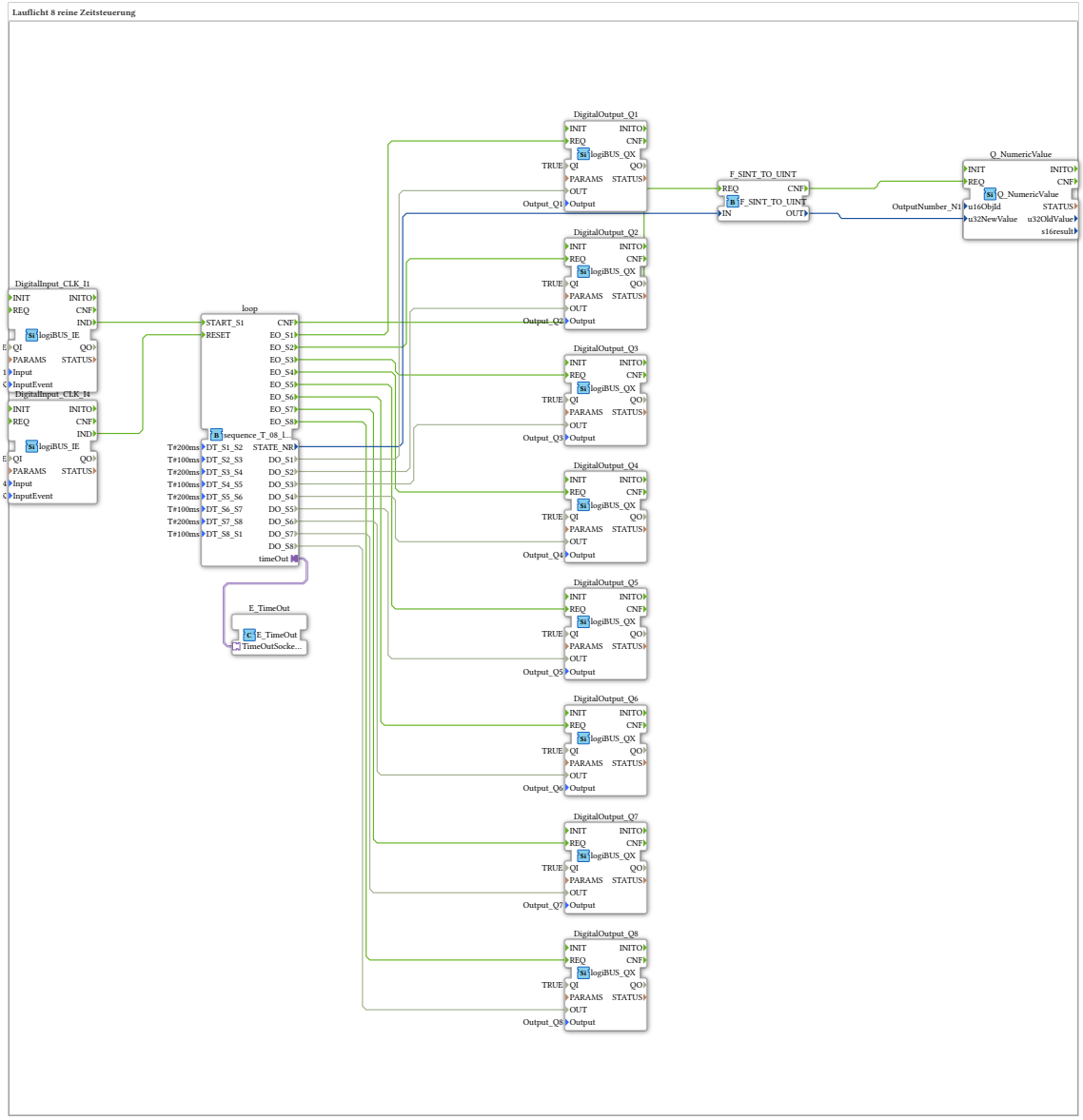
Diese Übung nutzt den Baustein `sequence_ET_04_loop`. Sobald die Sequenz durch Taster **I1** gestartet wurde, durchläuft sie permanent die Schritte 1 bis 4. Nach Abschluss von Schritt 4 springt sie automatisch wieder zu Schritt 1 zurück. Erst durch den Reset-Taster **I4** wird der Kreislauf unterbrochen und die Maschine gestoppt. Dies ist ideal für Dauerläufe oder periodische mechanische Bewegungen.

2.2.1.235 Uebung_038: Lauflicht 8 reine Zeitsteuerung

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_038`. Hier wird eine umfangreichere Schrittkette mit 8 Phasen realisiert.

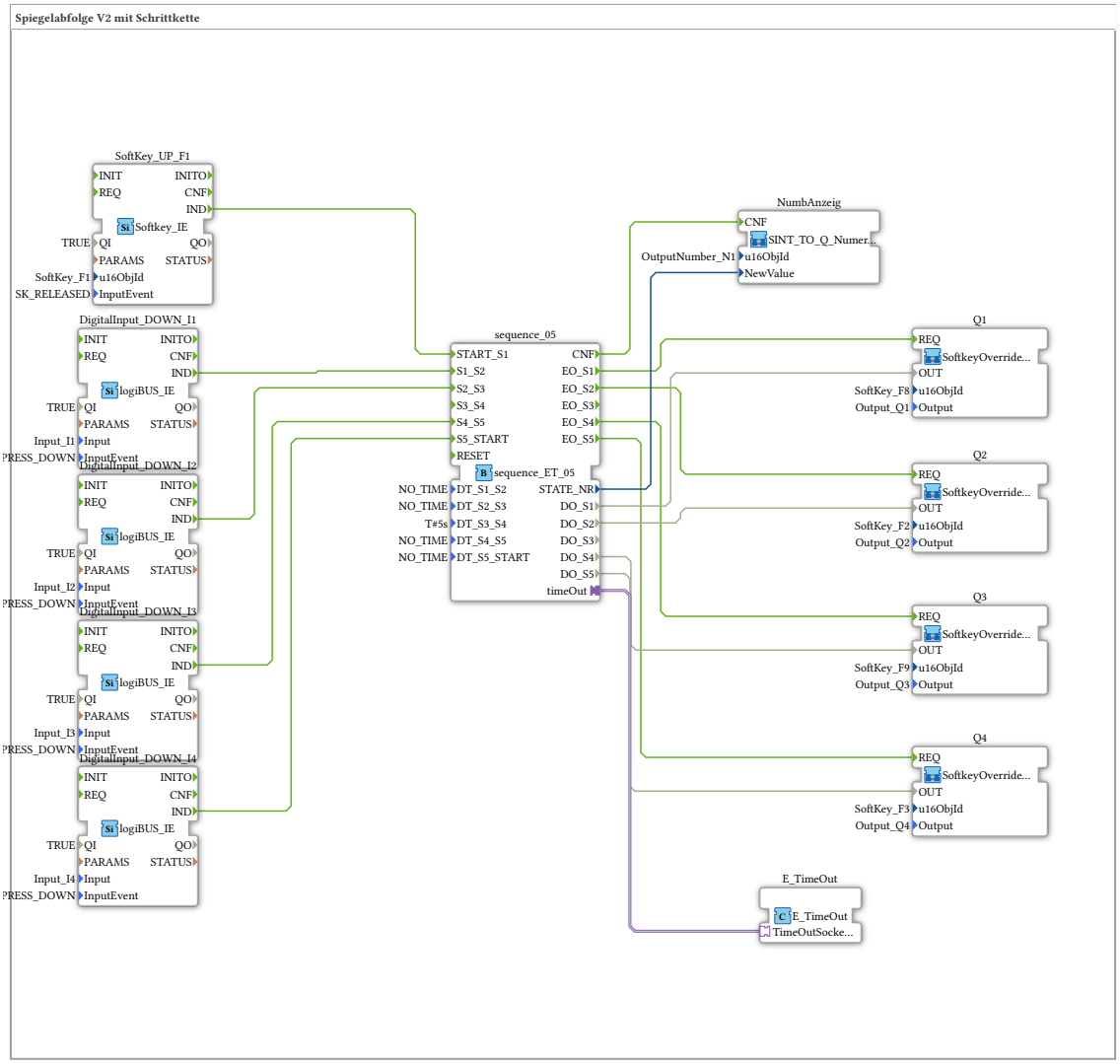
Übersicht

Unter Verwendung des Bausteins `sequence_T_08_loop` wird ein automatisches Lauflicht über 8 Ausgänge (`Q1` bis `Q8`) erzeugt. Die Übergangszeiten zwischen den Lampen sind individuell einstellbar (z.B. 200ms für die ungeraden, 100ms für die geraden Schritte). Das Programm demonstriert die Handhabung vieler paralleler Ausgänge und die numerische Rückmeldung des aktuellen Systemzustands an das Terminal.



2.2.1.236 Uebung_039: Spiegelabfolge V2 mit Schrittkette

NotebookLM



• – ○ • – ○ • – ○ •

Einleitung

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_039`. Diese Übung ist speziell auf die Ansteuerung von hydraulischen oder pneumatischen Wegeventilen zugeschnitten.

Ziel ist die Realisierung einer komplexen Spiegelabfolge. Im Gegensatz zu einfachen Zylindern müssen bei Wegeventilen oft Zustände gehalten werden (Mittelstellung gesperrt), was eine präzise zeitliche und ereignisbasierte Steuerung der Spulen erfordert.

Die Subapplikation `Uebung_039.SUB` nutzt einen 5-Schritt-Sequenzer (`sequence_ET_05`). Die Ansteuerung der Hardware erfolgt über vier Instanzen der wiederverwendbaren SubApp `MyLib::sys::SoftkeyOverride_TO_QX_BG`, die den jeweiligen Ventilzustand auch visuell auf dem ISO-BUS-Terminal durch Farbumschlag des zugehörigen Softkeys rückmelden.

Verwendete Funktionsbausteine (FBs)

- **SoftKey_UP_F1**: VT-Softkey als Start-Taster
  - **Typ**: `isobus::UT::io::Softkey::Softkey_IE`
  - **Parameter**: `QI = TRUE`, `u16ObjId = SoftKey_F1`, `InputEvent = SK_RELEASED`
  - **Erklärung**: Meldet das Loslassen von `SoftKey_F1` als Event `IND`.
- **DigitalInput_DOWN_I1** bis **DigitalInput_DOWN_I4**: logiBUS Eingänge mit Erkennung des Tastendrucks
  - **Typ**: `logiBUS::io::DI::logiBUS_IE`
  - **Parameter**: `QI = TRUE`, `Input = Input_I1 / Input_I2 / Input_I3 / Input_I4`, `InputEvent = BUTTON_PRESS_DOWN`

  - **Erklärung**: Jeder Eingang meldet das Herunterdrücken seines Tasters als Event `IND`.
- **sequence_05**: 5-Schritt-Sequenzer (Weiterschaltung per Ereignis oder Zeit)
  - **Typ**: `logiBUS::utils::sequence::combi::sequence_ET_05`
  - **Parameter**: `DT_S1_S2 = NO_TIME`, `DT_S2_S3 = NO_TIME`, `DT_S3_S4 = T#5s`, `DT_S4_S5 = NO_TIME`, `DT_S5_START = NO_TIME`
  - **Erklärung**: Schrittkette mit den Zuständen `START` (0) und `State_01` bis `State_05`. Jeder Übergang erfolgt über das zugehörige Ereignis (`START_S1`, `S1_S2`, ...) oder - sofern eine Zeit ungleich `NO_TIME` gesetzt ist - automatisch nach Ablauf der Zeit. Ausgänge: `STATE_NR` (mit `CNF`) sowie je Schritt `DO_Sx` mit `EO_Sx`. Nur `DT_S3_S4` ist mit 5 s belegt, alle anderen Übergänge warten auf ihr Ereignis.
- **E_TimeOut**: Zeitgeber für den Sequenzer
  - **Typ**: `iec61499::events::E_TimeOut`
  - **Parameter**: keine
  - **Erklärung**: Stellt über den Adapter `TimeOutSocket` die Timeout-Funktion für `sequence_05.timeOut` bereit.
- **NumbAnzeig**: Anzeige der Schrittnummer (SubApp-Instanz)
  - **Typ**: `MyLib::sys::SINT_TO_Q_NumericValue`
  - **Parameter**: `u16ObjId = OutputNumber_N1`
  - **Erklärung**: Wandelt den SINT-Wert `NewValue` (hier `STATE_NR`) in UINT und schreibt ihn per `Q_NumericValue` in das VT-Zahlenfeld `OutputNumber_N1`.
- **Q1**, **Q2**, **Q3**, **Q4**: Ausgang mit Handbedienung und Rückmeldung (SubApp-Instanzen)
  - **Typ**: `MyLib::sys::SoftkeyOverride_TO_QX_BG`
  - **Parameter**: `Q1`: `u16ObjId = SoftKey_F8`, `Output = Output_Q1`; `Q2`: `u16ObjId = SoftKey_F2`, `Output = Output_Q2`; `Q3`: `u16ObjId = SoftKey_F9`, `Output = Output_Q3`; `Q4`: `u16ObjId = SoftKey_F3`, `Output = Output_Q4`
  - **Erklärung**: Das Programmsignal `OUT` wird intern per `OR_2` mit dem gedrückten Softkey (`Softkey_IX`) ODER-verknüpft und auf den Digitalausgang (`logiBUS_QX`) gelegt; derselbe Zustand färbt über `GreenWhiteBackground1` den Hintergrund des Softkeys.

Programmablauf und Verbindungen

1. `SoftKey_UP_F1.IND` ⟶ `sequence_05.START_S1`: Loslassen von `SoftKey_F1` startet die Kette (Schritt 1).
2. `DigitalInput_DOWN_I1.IND` ⟶ `sequence_05.S1_S2`: `I1` schaltet von Schritt 1 nach Schritt 2.
3. `DigitalInput_DOWN_I2.IND` ⟶ `sequence_05.S2_S3`: `I2` schaltet von Schritt 2 nach Schritt 3.
4. Schritt 3 ⟶ Schritt 4: kein Taster, sondern automatisch nach `DT_S3_S4 = T#5s` (`sequence_05.timeOut` ↔ `E_TimeOut.TimeOutSocket`). Der Eingang `S3_S4` ist nicht verdrahtet.
5. `DigitalInput_DOWN_I3.IND` ⟶ `sequence_05.S4_S5`: `I3` schaltet von Schritt 4 nach Schritt 5.
6. `DigitalInput_DOWN_I4.IND` ⟶ `sequence_05.S5_START`: `I4` beendet die Kette (zurück zu `START`).
7. Ausgänge: `EO_S1/DO_S1` ⟶ `Q1.REQ/Q1.OUT`, `EO_S2/DO_S2` ⟶ `Q2`, `EO_S4/DO_S4` ⟶ `Q3`, `EO_S5/DO_S5` ⟶ `Q4`. Schritt 3 hat keinen Ausgang.
8. `sequence_05.STATE_NR` ⟶ `NumbAnzeig.NewValue` und `sequence_05.CNF` ⟶ `NumbAnzeig.CNF`: die aktuelle Schrittnummer wird im VT angezeigt.

Verhalten: Die Kette wird manuell durch die physischen Taster (`I1` bis `I4`) weitergeschaltet, wobei der zentrale Zeitschritt (5 s bei `DT_S3_S4`) eine automatische Sicherheits- oder Wartephase einfügt. Pro Schritt ist nur der jeweilige Ausgang aktiv (`Q1` in Schritt 1, `Q2` in Schritt 2, in Schritt 3 keiner, `Q3` in Schritt 4, `Q4` in Schritt 5).

Zusammenfassung

Die Übung zeigt die Kombination aus freier Bedienbarkeit (ereignisgesteuerte Übergänge per Taster) und erzwungenen Prozesszeiten (zeitgesteuerter Übergang von Schritt 3 nach 4) in einer Schrittkette für Wegeventile. Die Ausgänge werden über die generische SubApp `SoftkeyOverride_TO_QX_BG` angesteuert, sodass jede Spule zusätzlich per Softkey von Hand betätigt werden kann und der Zustand auf dem Terminal sichtbar ist.

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

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

2.2.1.237 Uebung_039_sub_NumbAnzeig

Hinweis

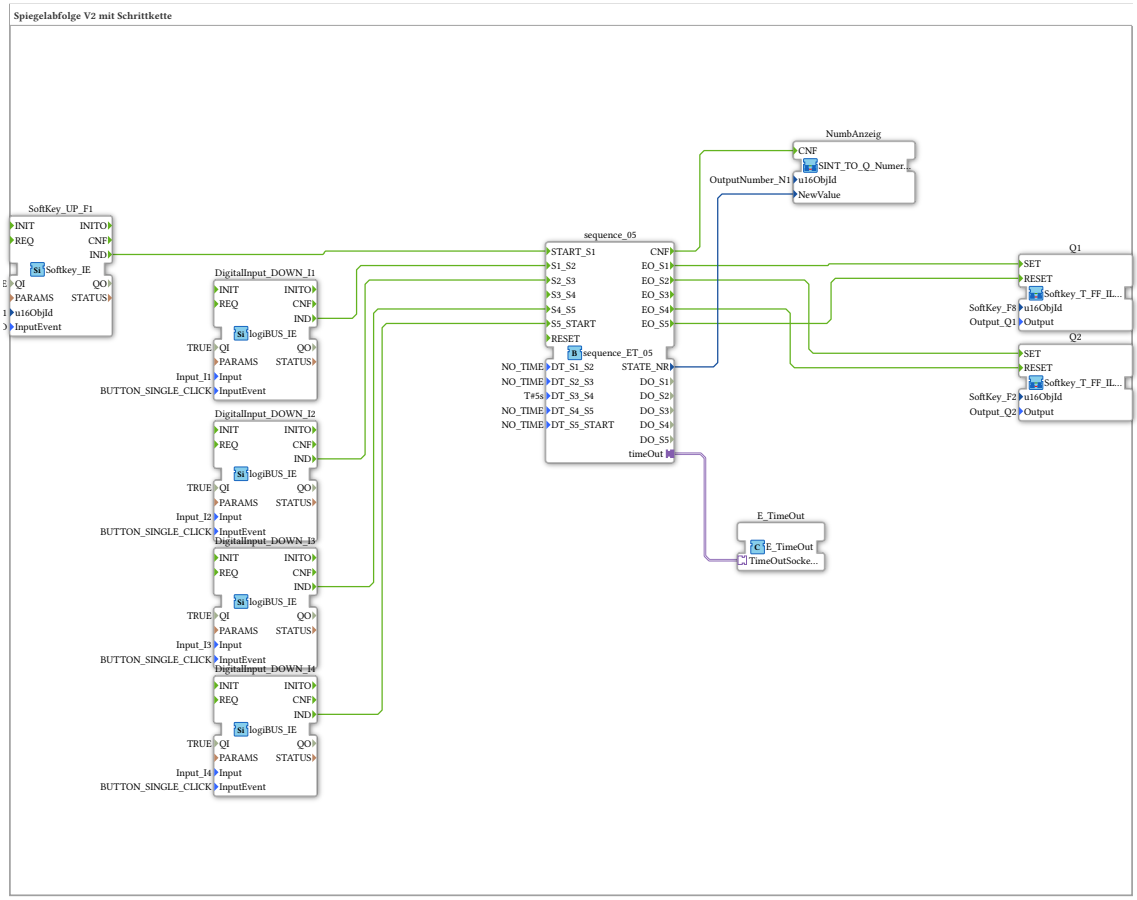
Diese Übung existiert nicht mehr als eigene Übung. Der Baustein wurde in die Bibliothek verschoben und heißt jetzt SINT_TO_Q_NumericValue. Beschreibung, Schnittstelle und Netzwerk stehen dort.

2.2.1.238 Uebung_039_sub_Outputs

Hinweis

Diese Übung existiert nicht mehr als eigene Übung. Der Baustein wurde in die Bibliothek verschoben und heißt jetzt SoftkeyOverride_TO_QX_BG. Beschreibung, Schnittstelle und Netzwerk stehen dort.

2.2.1.239 Uebung_039a: Spiegelabfolge V2 mit Schrittkette



• — ○ • — ○ • — ○ •



Einleitung

Diese Übung ist eine Spezialisierung der Ventilsteuerung für Systeme mit 3/2-Wege-Ventilen (z. B. hydraulische Ringsysteme wie bei Claas). Die Schrittkette `sequence_ET_05` verwaltet den Ablauf der Zylinderbewegungen. Die Ansteuerung der physischen Ausgänge `Q1` und `Q2` erfolgt über je eine Instanz des Bausteins `Softkey_T_FF_ILOCK_TO_QX_BG`, die zusätzlich ein direktes visuelles Feedback auf dem ISOBUS-Softkey liefert. Die aktuelle Schrittnummer wird auf einem VT-Zahlenfeld angezeigt. Dies demonstriert die Anpassbarkeit der logiBUS-Sequenz-Bausteine an verschiedene hydraulische Verschaltungskonzepte.

Verwendete Funktionsbausteine (FBs)

- **SoftKey_UP_F1**: Softkey-Eingang für den Start

  - **Typ**: `isobus::UT::io::Softkey::Softkey_IE`
  - **Parameter**: `QI = TRUE`, `u16ObjId = SoftKey_F1`, `InputEvent = SK_RELEASED`
  - **Erklärung**: Meldet das Loslassen des Softkeys `SoftKey_F1` als Ereignis `IND`.
- **DigitalInput_DOWN_I1** … **DigitalInput_DOWN_I4**: Klick-Eingänge für die manuellen Schrittübergänge
  - **Typ**: `logiBUS::io::DI::logiBUS_IE`
  - **Parameter**: `QI = TRUE`, `Input = Input_I1` bzw. `Input_I2`, `Input_I3`, `Input_I4`, jeweils `InputEvent = BUTTON_SINGLE_CLICK`
  - **Erklärung**: Melden einen Einzelklick auf `I1` bis `I4` als Ereignis `IND`.
- **sequence_05**: Schrittkette
  - **Typ**: `logiBUS::utils::sequence::combi::sequence_ET_05`
  - **Parameter**: `DT_S1_S2 = NO_TIME`, `DT_S2_S3 = NO_TIME`, `DT_S3_S4 = T#5s`, `DT_S4_S5 = NO_TIME`, `DT_S5_START = NO_TIME`
  - **Erklärung**: Fünf Schritte mit Start-Zustand. Jeder Übergang erfolgt ereignisgesteuert oder zeitgesteuert; `NO_TIME` schaltet die Zeitsteuerung ab. Hier läuft nur der Übergang von Schritt 3 nach Schritt 4 automatisch nach 5 s, alle anderen Übergänge sind manuell. Der Baustein liefert beim Eintritt in Schritt n das Ereignis `EO_Sn` und bei jedem Wechsel `CNF` mit der Schrittnummer `STATE_NR`.
- **E_TimeOut**: Zeitgeber der Schrittkette
  - **Typ**: `iec61499::events::E_TimeOut`
  - **Parameter**: keine
  - **Erklärung**: Stellt über den Adapter `TimeOutSocket` die Zeitüberwachung für die zeitgesteuerten Übergänge von `sequence_05` bereit.
- **Q1**: Ausgang `Q1` mit Softkey-Feedback
  - **Typ**: `MyLib::sys::Softkey_T_FF_ILOCK_TO_QX_BG` (SubApp-Instanz)
  - **Parameter**: `u16ObjId = SoftKey_F8`, `Output = Output_Q1`
  - **Erklärung**: Digitalausgang als Toggle-Flip-Flop: Ein Loslassen des Softkeys `SoftKey_F8` kippt den Ausgang, zusätzlich können die Ereignisse `SET` und `RESET` den Ausgang von außen setzen bzw. zurücksetzen (Verriegelung). Die Hintergrundfarbe des Softkeys spiegelt den Zustand.
- **Q2**: Ausgang `Q2` mit Softkey-Feedback
  - **Typ**: `MyLib::sys::Softkey_T_FF_ILOCK_TO_QX_BG` (SubApp-Instanz)
  - **Parameter**: `u16ObjId = SoftKey_F2`, `Output = Output_Q2`
  - **Erklärung**: Wie `Q1`, jedoch für den Softkey `SoftKey_F2` und den Ausgang `Q2`.
- **NumbAnzeig**: Anzeige der Schrittnummer
  - **Typ**: `MyLib::sys::SINT_TO_Q_NumericValue` (SubApp-Instanz)
  - **Parameter**: `u16ObjId = OutputNumber_N1`
  - **Erklärung**: Wandelt den SINT-Wert `NewValue` in UINT um und zeigt ihn auf dem VT-Zahlenfeld `OutputNumber_N1` an.

Programmablauf und Verbindungen

1. `SoftKey_UP_F1.IND` ⟶ `sequence_05.START_S1` (EventConnection): Das Loslassen von `SoftKey_F1` startet die Schrittkette in Schritt 1.
2. `DigitalInput_DOWN_I1.IND` ⟶ `sequence_05.S1_S2`, `DigitalInput_DOWN_I2.IND` ⟶ `sequence_05.S2_S3`, `DigitalInput_DOWN_I3.IND` ⟶ `sequence_05.S4_S5` und `DigitalInput_DOWN_I4.IND` ⟶ `sequence_05.S5_START`: Ein Klick auf `I1`, `I2`, `I3` bzw. `I4` schaltet manuell von Schritt 1 nach 2, von 2 nach 3, von 4 nach 5 bzw. von 5 zurück zum Start. Der Übergang von Schritt 3 nach 4 ist nicht verdrahtet und läuft zeitgesteuert nach `DT_S3_S4 = T#5s`.
3. `sequence_05.timeOut` ⟶ `E_TimeOut.TimeOutSocket` (AdapterConnection): Die Zeitsteuerung der Schrittkette.
4. `sequence_05.EO_S1` ⟶ `Q1.SET`: In Schritt 1 wird `Q1` gesetzt.
5. `sequence_05.EO_S2` ⟶ `Q2.SET`: In Schritt 2 wird `Q2` gesetzt.
6. `sequence_05.EO_S4` ⟶ `Q2.RESET`: In Schritt 4 wird `Q2` zurückgesetzt.
7. `sequence_05.EO_S5` ⟶ `Q1.RESET`: In Schritt 5 wird `Q1` zurückgesetzt.
8. `sequence_05.CNF` ⟶ `NumbAnzeig.CNF` (EventConnection) und `sequence_05.STATE_NR` ⟶ `NumbAnzeig.NewValue` (DataConnection): Bei jedem Schrittwechsel wird die Schrittnummer auf dem VT angezeigt.

Verhalten (Spiegelabfolge): `Q1` wird in Schritt 1 gesetzt und erst in Schritt 5 zurückgesetzt; `Q2` wird in Schritt 2 gesetzt und in Schritt 4 zurückgesetzt. Die Ausgänge werden also in spiegelbildlicher Reihenfolge ein- und ausgeschaltet.

Zusammenfassung

Die Übung zeigt, wie eine fünfstufige Schrittkette (`sequence_ET_05`) zwei Ausgänge in einer Spiegelabfolge setzt und zurücksetzt, wobei Übergänge teils manuell über Klick-Eingänge und teils zeitgesteuert erfolgen. Die Ausgangsbausteine geben auf den Softkeys visuelles Feedback, und die aktuelle Schrittnummer wird auf dem VT angezeigt.

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

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

2.2.1.240 Uebung_039a_sub_Outputs

Hinweis

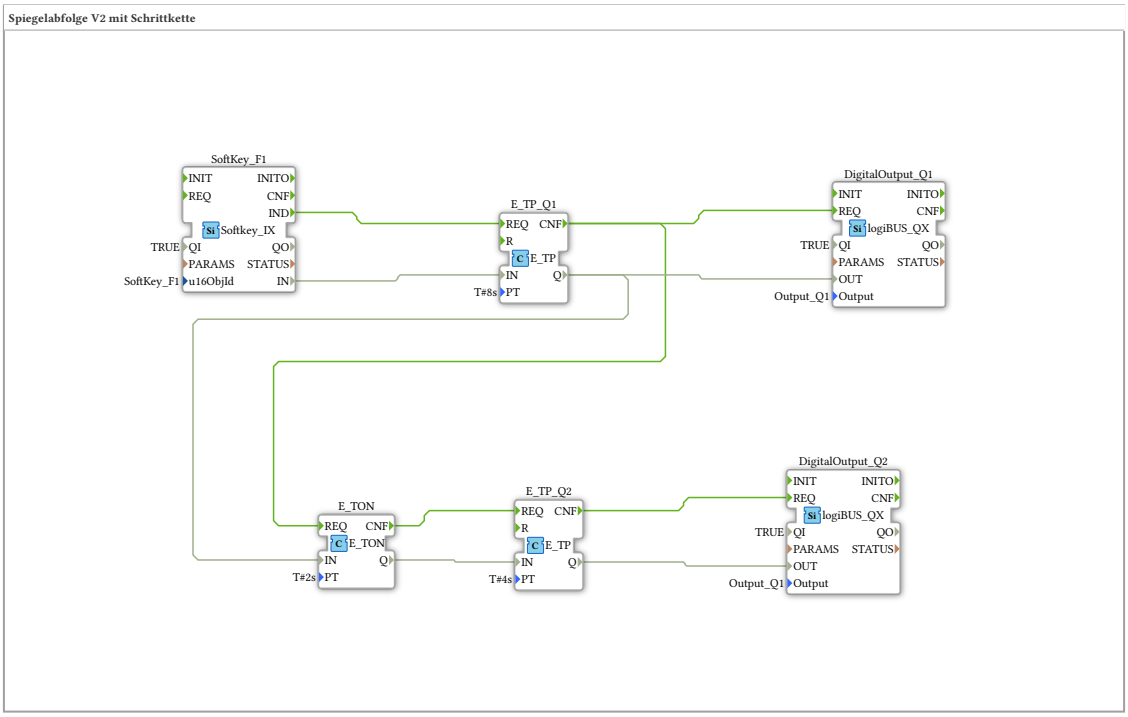
Diese Übung existiert nicht mehr als eigene Übung. Der Baustein wurde in die Bibliothek verschoben und heißt jetzt Softkey_T_FF_ILOCK_TO_QX_BG. Beschreibung, Schnittstelle und Netzwerk stehen dort.

2.2.1.241 Uebung_039b: Spiegelabfolge V2 mit Schrittkette



Übersicht

In dieser Übung wird eine zeitgesteuerte Ventil-Abfolge unter Verwendung von Impulsgebern (`E_TP`) realisiert.



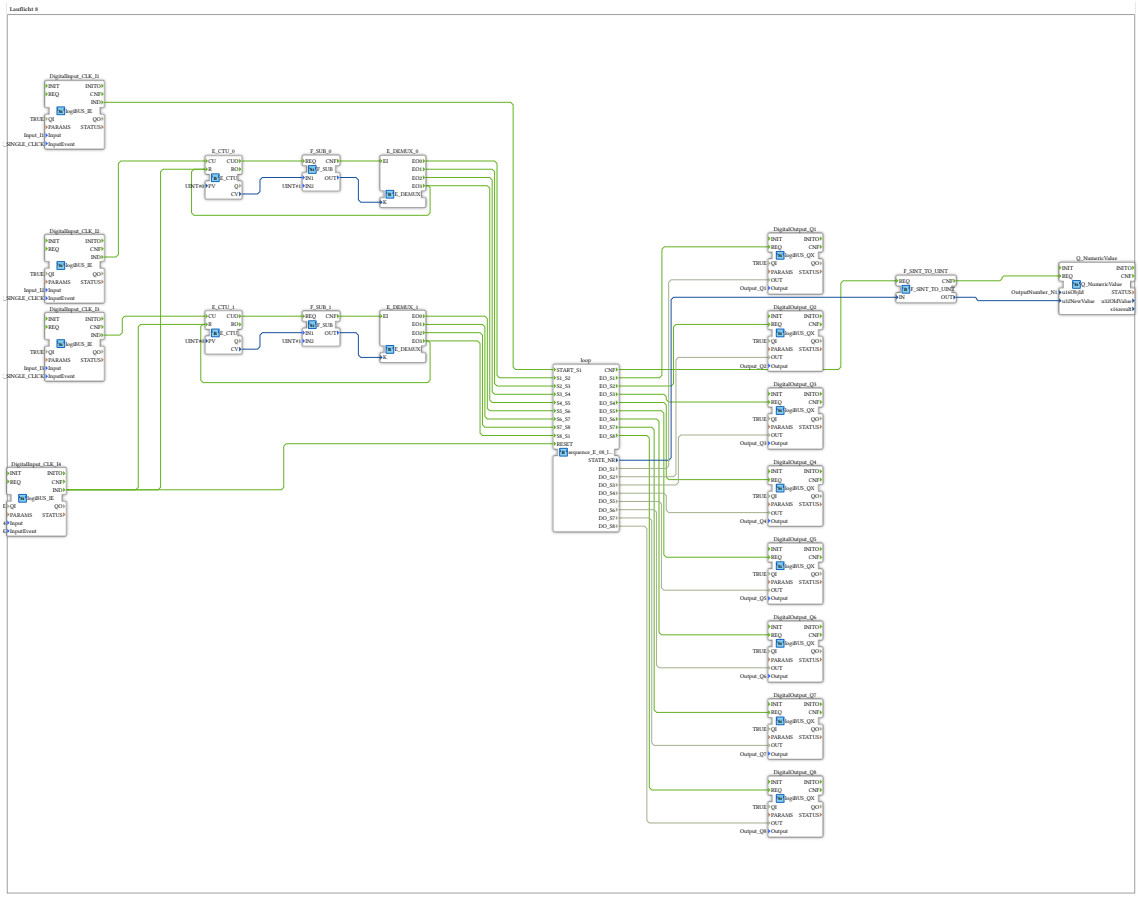
Ein Klick auf den Softkey **F1** startet eine Kette von Ereignissen:

1. Ventil **Q1** wird für 8 Sekunden geöffnet.
2. Nach einer Verzögerung von 2 Sekunden (`E_TON`) wird Ventil **Q2** für 4 Sekunden hinzugeschaltet. Dies ermöglicht die Programmierung von festen hydraulischen Funktionszyklen (z.B. „Ballenauswurf“), bei denen mehrere Aktoren mit exaktem Zeitversatz arbeiten müssen.

2.2.1.242 Uebung_040: Lauflicht 8

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_040`. Hier wird eine raffinierte Methode gezeigt, um eine 8-stufige Schrittkette mit nur wenigen Tastern manuell durchzuschalten.

📺 Video



- From 1,400 errors to zero

🎧 Podcast

- Von 1400 Fehlern zum sauberen Code: Die Migration der „Getreidehacke“ auf Eclipse 4diac™ 3.0 und die Macht der AX-Adapter
- 400 Millionen Tonnen Agrar-Logistik digitalisieren
- Schlüter 1500 Spezial: Turbo-Giftigkeit, 40 Jahre und die Seele eines Kraftprotzes

---

Ziel der Übung

Kombination von Ereignis-Zählern (`E_CTU`) und Ereignis-Demultiplexern (`E_DEMUX`) zur Steuerung einer Ablaufkette.

Beschreibung und Komponenten

Die Subapplikation `Uebung_040.SUB` nutzt zwei Zähler-Zweige, um die Ereignis-Eingänge des Sequenzers `sequence_E_08_loop` zu bedienen.

Funktionsweise

1. **Start**: Taster **I1** setzt die Kette auf Anfang (Schritt 1).

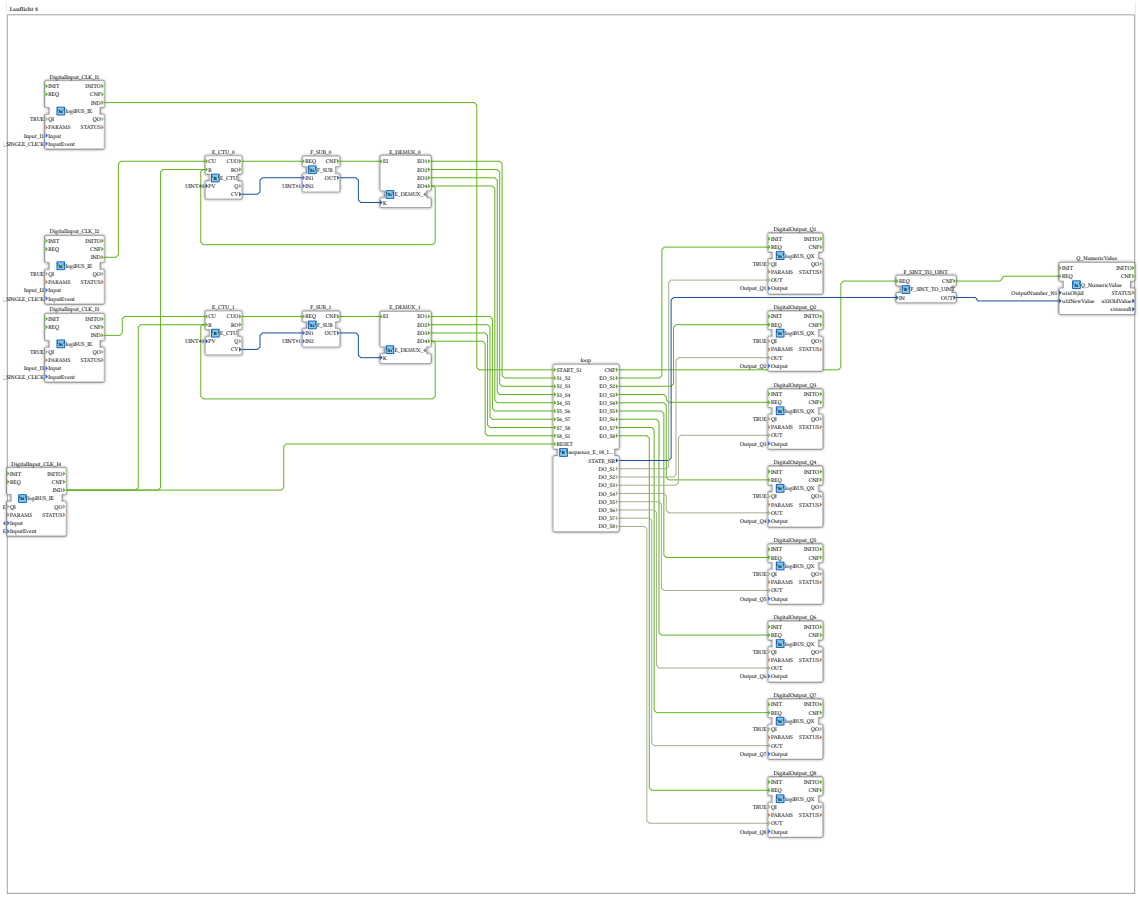
2. **Steppen 1-4**: Jeder Klick auf Taster **I2** erhöht den ersten Zähler. Der Demultiplexer leitet das Klick-Event nacheinander an die Eingänge `S1_S2`, `S2_S3` etc. weiter. Der Nutzer „klickt“ sich also durch die ersten vier Phasen.
3. **Steppen 5-8**: Taster **I3** übernimmt analog die Steuerung für die zweite Hälfte der Kette.
4. **Zähler-Reset**: Sobald ein Demultiplexer den letzten Ausgang erreicht hat, setzt er seinen zugehörigen Zähler automatisch wieder auf Null zurück.

Dies ist eine sehr effiziente Methode, um komplexe manuelle Abläufe auf kleinstem Raum (wenige Bedienelemente) abzubilden.

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

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

2.2.1.243 Uebung_040_2: Lauflicht 8



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung implementiert ein **8-Kanal-Lauflicht**, welches manuell über Taster gesteuert wird. Im Gegensatz zu einem automatisch ablaufenden Lauflicht, wird hier der Fortschritt der Sequenz durch Benutzerinteraktion bestimmt. Das System ist in zwei Blöcke aufgeteilt, die jeweils 4 Schritte steuern, und beinhaltet eine visuelle Rückmeldung über den aktuellen Status sowie eine numerische Anzeige des aktiven Schritts.

Verwendete Funktionsbausteine (FBs)

In dieser Übung werden verschiedene Standard- und Spezialbausteine verwendet, um die Logik, die Ein-/Ausgabe und die Sequenzsteuerung zu realisieren.

Sub-Bausteine: sequence_E_08_loop

Dies ist der zentrale Baustein zur Steuerung der Zustände des Lauflichts.

- **Typ**: `logiBUS::utils::sequence::event::sequence_E_08_loop`
- **Verwendete interne FBs**: (Angenommen basierend auf der Schnittstelle)
  - **Interne Logik**: Zustandsautomat (State Machine)
- **Funktionsweise**: Dieser Baustein verwaltet 8 sequentielle Zustände (S1 bis S8). Er verfügt über Ereigniseingänge, um spezifische Übergänge auszulösen (z.B. `S1_S2` für den Wechsel von Schritt 1 zu 2).
  - **Eingänge**: `START_S1` (Initialisierung), `RESET` (Zurücksetzen), sowie diverse Übergangstrigger (`S1_S2`, `S2_S3`, etc.).
  - **Ausgänge**: Für jeden Zustand gibt es ein Ereignis (`EO_S1`..`EO_S8`) und ein Datensignal (`DO_S1`..`DO_S8`), welche die physikalischen Ausgänge steuern. Zusätzlich wird die aktuelle Zustandsnummer (`STATE_NR`) ausgegeben.

Sub-Bausteine: Logik-Cluster (Zähler & Demultiplexer)

Um die manuellen Tasterdrücke in die korrekten Zustandsübergänge umzuwandeln, wird eine Kombination aus Zählern, Subtrahierern und Demultiplexern verwendet. Dies kommt zweimal vor (für die erste und zweite Hälfte der Sequenz).

- **Typ**: Kombination aus Standard-IEC61499/61131 Bausteinen
- **Verwendete interne FBs**:
  - **E_CTU** (`iec61499::events::E_CTU`): Aufwärtszähler. Zählt die Tasterdrücke.
  - **F_SUB** (`iec61131::arithmetic::F_SUB`): Subtrahierer. Zieht vom Zählerwert 1 ab, um einen 0-basierten Index für den Demultiplexer zu erhalten.
  - **E_DEMUX_4** (`iec61499::events::E_DEMUX_4`): Demultiplexer. Leitet das Eingangssignal basierend auf dem Index `K` an einen von 4 Ausgängen weiter.
- **Funktionsweise**: Ein Tasterdruck erhöht den Zähler. Der Wert wird angepasst (Schritt 1 wird zu Index 0) und steuert den Demultiplexer. Dieser feuert das entsprechende Event, um den `sequence_E_08_loop` Baustein in den nächsten Zustand zu schalten.

Weitere Bausteine

- **logiBUS_QX (DigitalOutput_Q1 - Q8)**: Repräsentieren die 8 Lampen/LEDs des Lauflichts.
- **logiBUS_IE (DigitalInput_CLK_I1 - I4)**: Repräsentieren die Eingabetaster.
  - `I1`: Start / Initialisierung.
  - `I2`: Weiterschalten Schritte 1-4.
  - `I3`: Weiterschalten Schritte 5-8.
  - `I4`: Reset.
- **Q_NumericValue**: Zeigt den aktuellen Schritt (1-8) auf dem Display an.
- **F_SINT_TO_UINT**: Konvertiert den Datentyp der Zustandsnummer für die Anzeige.

Programmablauf und Verbindungen

Das Programm ist darauf ausgelegt, eine geführte Sequenz von 8 Schritten abzubilden.

1. **Start und Reset**:
   - Über den Taster **I1** (`START_S1`) wird das Lauflicht gestartet (Zustand 1 aktiv, Q1 leuchtet).
   - Über den Taster **I4** (`RESET`) kann das System jederzeit zurückgesetzt werden. Dies setzt sowohl den Sequenz-Baustein als auch die Zähler (`E_CTU_0`, `E_CTU_1`) zurück.
2. **Sequenzsteuerung Teil 1 (Schritte 1 bis 5)**:
   - Der Taster **I2** ist für die ersten vier Übergänge zuständig.
   - Bei jedem Klick zählt `E_CTU_0` hoch.
   - Der Demultiplexer `E_DEMUX_0` verteilt diese Ereignisse nacheinander an die Eingänge `S1_S2`, `S2_S3`, `S3_S4` und `S4_S5` des Loop-Bausteins.
   - Wenn der 4. Schritt erreicht ist (Ausgang EO4 des Demux), wird der Zähler automatisch zurückgesetzt.
3. **Sequenzsteuerung Teil 2 (Schritte 5 bis 1)**:
   - Der Taster **I3** übernimmt die Steuerung für die zweite Hälfte.

- Er steuert über `E_CTU_1` und `E_DEMUX_1` die Übergänge `S5_S6`, `S6_S7`, `S7_S8` und schließlich `S8_S1` (zurück zum Anfang).

4. **Ausgabe**:
   - Der `sequence_E_08_loop` Baustein aktiviert je nach internem Zustand genau einen der Ausgänge **Q1 bis Q8**.
   - Gleichzeitig wird die aktuelle Zustandsnummer (`STATE_NR`) über den Konverter `F_SINT_TO_UINT` an das Display-Element `OutputNumber_N1` gesendet, um dem Benutzer den aktuellen Schritt anzuzeigen.

Lernziele:

- Verständnis von Zustandsautomaten (Sequencern).
- Nutzung von Zählern und Demultiplexern zur Ereignissteuerung.
- Aufteilung von Steuerungsaufgaben auf verschiedene Eingänge.
- Umgang mit Datentypkonvertierung.

Zusammenfassung

Die Übung `Uebung_040_2` demonstriert ein komplexes, manuell getaktetes Lauflicht. Durch die Verwendung eines dedizierten Sequenz-Bausteins (`sequence_E_08_loop`) wird die Logik der Zustände sauber gekapselt, während die externe Beschaltung mit Zählern und Demultiplexern eine flexible Eingabesteuerung über mehrere Taster ermöglicht. Das Ergebnis ist eine robust steuerbare Lichtsequenz mit visueller Statusanzeige.

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

- 🌐 E_CTU Event Counter Baustein auf ms-muc-docs.de

2.2.1.244 Uebung_041: Lauflicht 8 mit einer Taste



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_041`. Hier wird die manuelle Steuerung einer 8-stufigen Schrittkette auf einen einzigen Taster reduziert.

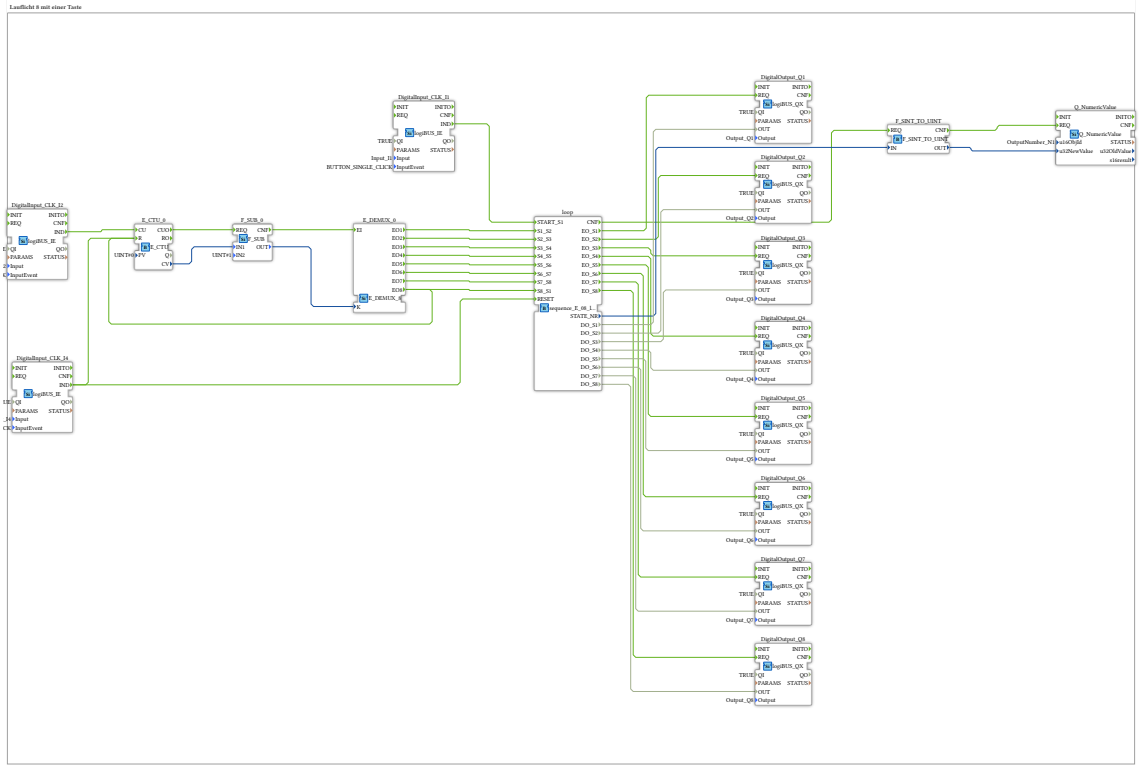
Ziel der Übung

Optimierung der Bedienlogik aus Übung 040. Es wird gezeigt, wie man durch die Kombination von Zähler (`E_CTU`) und Demultiplexer (`E_DEMUX_8`) alle Phasen einer Schrittkette mit nur einer einzigen Taste nacheinander durchschalten kann.

Beschreibung und Komponenten

In `Uebung_041.SUB` wird ein zentraler Ereignispfad genutzt, um den Sequenzer `sequence_E_08_loop` anzusteuern.

Funktionsbausteine (FBs)



- **`I1` (Start)**: Setzt die Sequenz auf den ersten Schritt.
- **`I2` (Step)**: Der einzige Taster zum Weiterschalten.
- **`E_CTU_0`**: Zählt die Klicks auf `I2`.
- **`E_DEMUX_0`**: Leitet das Klick-Ereignis basierend auf dem Zählerstand an den passenden Transitions-Eingang der Schrittkette weiter.
- **`I4` (Reset)**: Löscht sowohl die Schrittkette als auch den Zähler.

Funktionsweise

1. **Initialisierung**: Ein Klick auf **I1** startet das Lauflicht bei `Q1`.
2. **Manueller Durchlauf**: Jeder Klick auf **I2** erhöht den internen Zähler. Der Demultiplexer sorgt dafür, dass das erste Event an `S1_S2` geht, das zweite an `S2_S3` und so weiter.
3. **Überlauf**: Nach dem
4. Schritt setzt sich die Logik automatisch zurück und beginnt (beim nächsten Klick) wieder von vorn.

Dies ermöglicht eine vollständige Prozesskontrolle mit minimaler Hardware-Anforderung.

Anwendungsbeispiel

Sequenzielle Menüführung: Ein einziger Knopf am Joystick dient zum Durchblättern von 8 verschiedenen Betriebsmodi. Jede Betätigung schaltet eine Stufe weiter und aktiviert den entsprechenden Aktor oder Parameter-Satz.

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

- 🌐 E_CTU Event Counter Baustein auf ms-muc-docs.de

2.2.1.245 Uebung_042: Scaling Function Block Testing



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_042`. Hier wird die mathematische Umrechnung von Wertebereichen demonstriert.

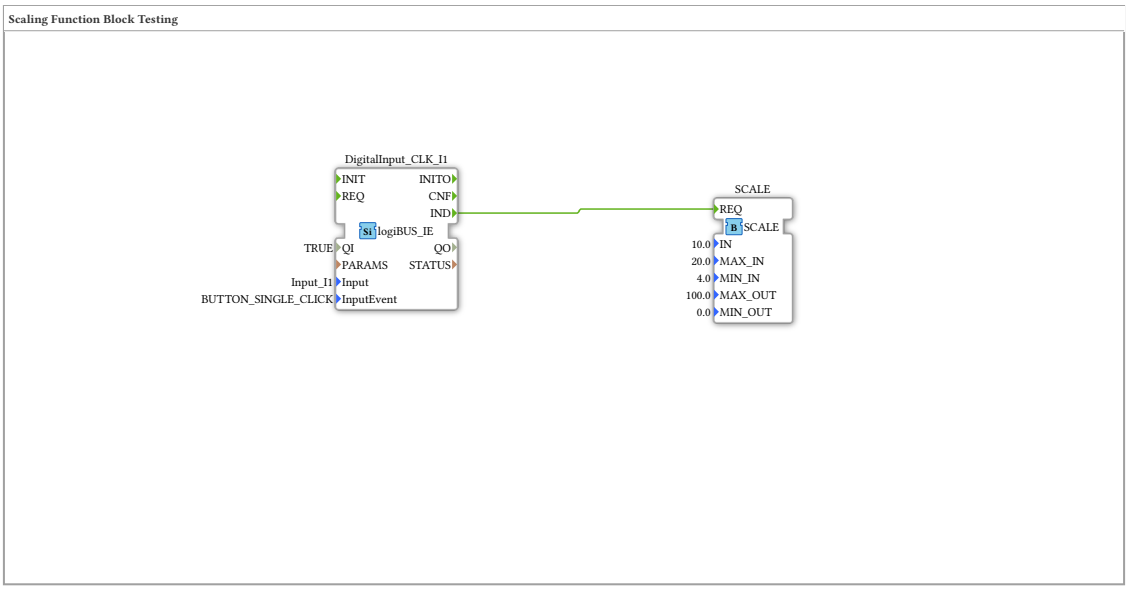
Ziel der Übung

Verwendung des Bausteins `SCALE`. In der Automatisierungstechnik müssen Rohwerte (z.B. 4-20 mA) oft in physikalische Größen (z.B. 0-10 Bar) umgerechnet werden. Der Scale-Baustein übernimmt diese lineare Abbildung.

Beschreibung und Komponenten

In `Uebung_042.SUB` wird ein Test-Szenario für den Skalierungs-Baustein aufgebaut.

Funktionsbausteine (FBs)



- **`SCALE`**: Der Umrechnungs-Baustein.
- **Parameter**:
  - `MIN_IN` / `MAX_IN`: Der Quell-Bereich (hier 4.0 bis 20.0).
  - `MIN_OUT` / `MAX_OUT`: Der Ziel-Bereich (hier 0.0 bis 100.0).
  - `IN`: Der aktuelle Eingangswert (hier fest auf 10.0 gesetzt).

Funktionsweise

Sobald das Ereignis `REQ` (hier durch Taster **I1** ausgelöst) eintrifft, berechnet der Baustein die Position des Eingangswerts im Quell-Bereich und bildet diese proportional auf den Ziel-Bereich ab. Bei `IN = 10.0` (genau in der Mitte zwischen 4 und 20 ist es nicht ganz, aber mathematisch definiert) liefert der Baustein das entsprechende Ergebnis am Ausgang.

Anwendungsbeispiel

Sensorkalibrierung: Ein Drucksensor liefert Werte zwischen 400 (Vakuum) und 2000 (Maximaldruck). Für die Anzeige am Terminal soll dies als 0% bis 100% dargestellt werden. Der `SCALE`-Baustein übernimmt diese Aufgabe, sodass die Logik immer mit intuitiven Prozentwerten arbeiten kann.

2.2.1.246 Uebung_043: Scaling Function Block with limits Testing



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_043`. Dies ist eine Erweiterung der Skalierung um Sicherheitsgrenzen.

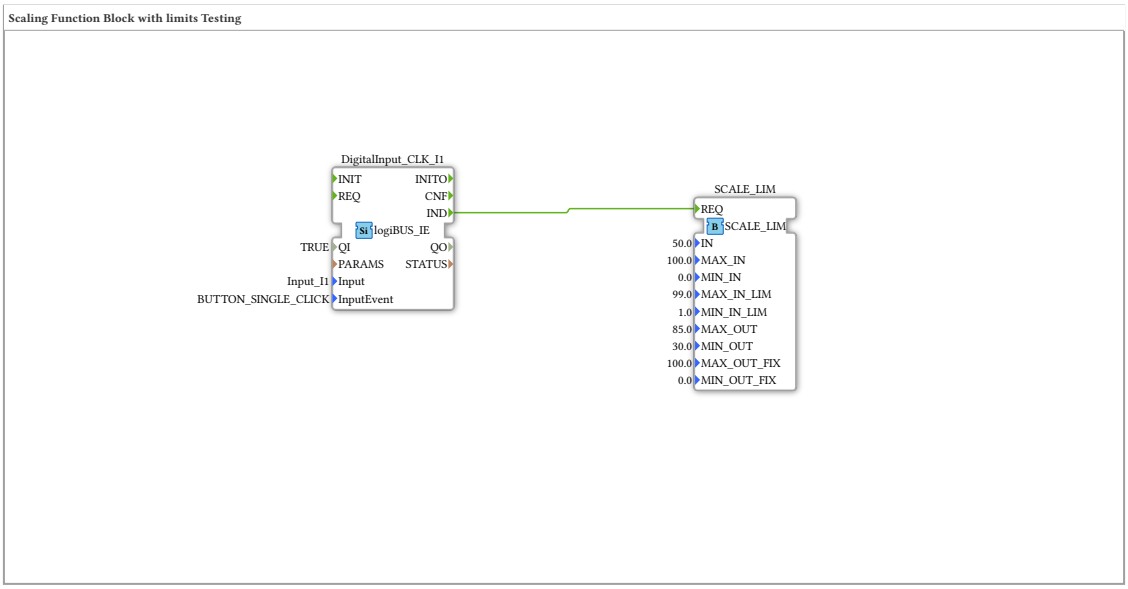
Ziel der Übung

Verwendung des Bausteins `SCALE_LIM`. Im Gegensatz zum einfachen `SCALE` bietet dieser Baustein zusätzliche Parameter, um das Ergebnis nach oben und unten zu begrenzen (Limiting), selbst wenn der Eingangswert den definierten Bereich verlässt.

Beschreibung und Komponenten

In `Uebung_043.SUB` wird ein hochkomplexer Skalierungs-Szenario mit fixen Grenzen aufgebaut.

Funktionsbausteine (FBs)



- **`SCALE_LIM`**: Skalierung mit Sättigung.
- **Parameter**:
  - `MIN_IN_LIM` / `MAX_IN_LIM`: Definieren den Bereich, in dem der Eingangswert „gültig“ ist.
  - `MIN_OUT_FIX` / `MAX_OUT_FIX`: Harte Grenzwerte für den Ausgang. Egal was berechnet wird, der Ausgang wird diese Werte niemals unter- oder überschreiten.

Anwendungsbeispiel

Überlaufschutz bei der Ventilsteuerung: Ein Regler berechnet die Öffnung eines Ventils basierend auf der Temperatur. Auch wenn der Regler aufgrund einer extremen Störung „150%“ anfordert, sorgt `SCALE_LIM` dafür, dass der reale Ausgangswert bei 100% gekappt wird, um die Hardware nicht zu beschädigen. Ebenso kann eine Mindestöffnung (z.B. 5% zur Kühlung) fest als Untergrenze hinterlegt werden.

2.2.1.247 Uebung_044: Reine Zeitsteuerung 5-Kanal

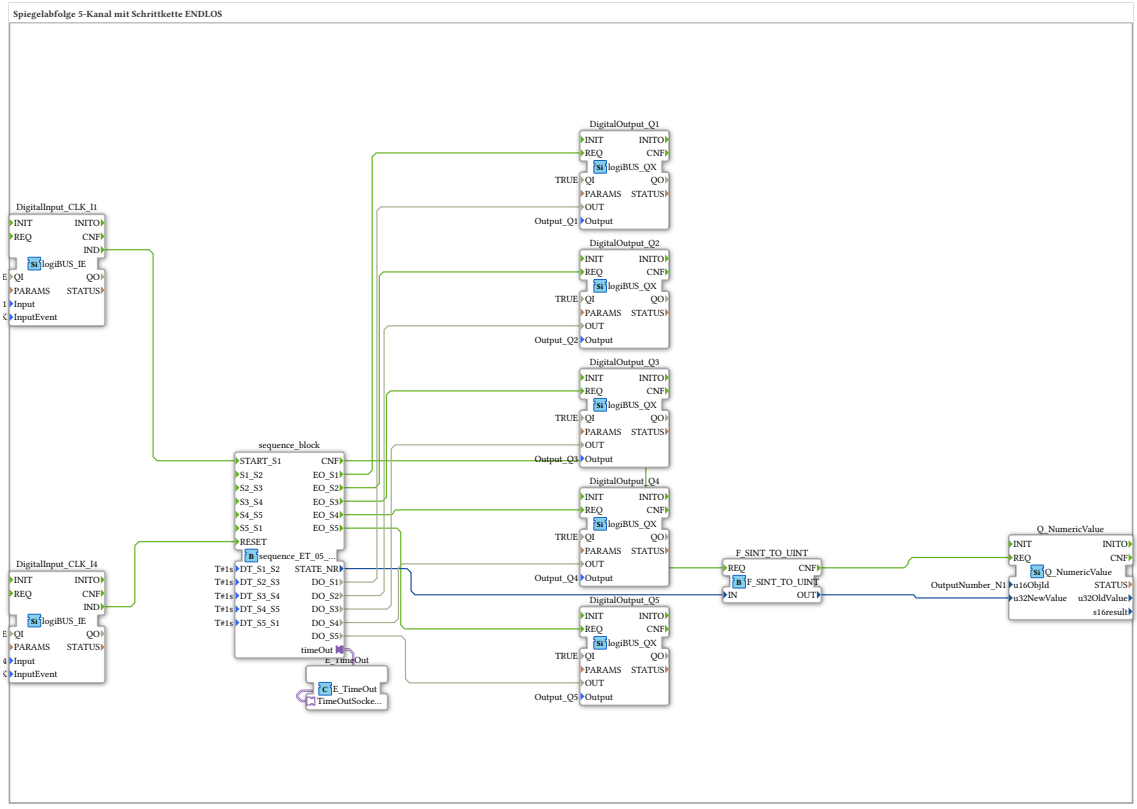
Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_044`. Wir bauen eine rein zeitgesteuerte Schrittkette.

Ziel der Übung

Realisierung einer automatischen Abfolge von 5 Ausgängen (Q1, Q2, Q3, Q4, Q5), bei der jeder Schritt ausschließlich nach Ablauf einer festen Wartezeit (`DT_S1_S2` usw., je `T#1s`) zum nächsten wechselt - ganz ohne weitere Tasterbetätigung während des Ablaufs.

Beschreibung und Komponenten

Die Subapplikation `Uebung_044` nutzt den Sequenzer `sequence_ET_05_loop`, der 5 Zustände (`S1` bis `S5`) rein zeitgesteuert durchläuft.



- **`sequence_ET_05_loop`**: Verwaltet die 5 Schritte, jeder Übergang erfolgt nach Ablauf der jeweiligen `DT_Sx_Sy`-Parameter.
- **`E_TimeOut`**: Überwacht die Sequenz als Watchdog.
- **`Q_NumericValue`**: Zeigt die aktuelle Schrittnummer auf dem ISOBUS-Terminal an.

Funktionsweise

1. Start durch Taster `I1` -> Sequenz springt zu `S1`.
2. Jeder Ausgang (`Q1, Q2, Q3, Q4, Q5`) wird für die eingestellte Zeit aktiv geschaltet, bevor automatisch zum nächsten Schritt gewechselt wird.
3. Nach dem letzten Schritt kehrt die Sequenz automatisch zu S1 zurück (ENDLOS).
4. Reset durch Taster `I4` -> Alles aus, Sequenz zurück in den Grundzustand.

2.2.1.248 Uebung_045: Ablaufsteuerung 8-Kanal (Event & Zeit)

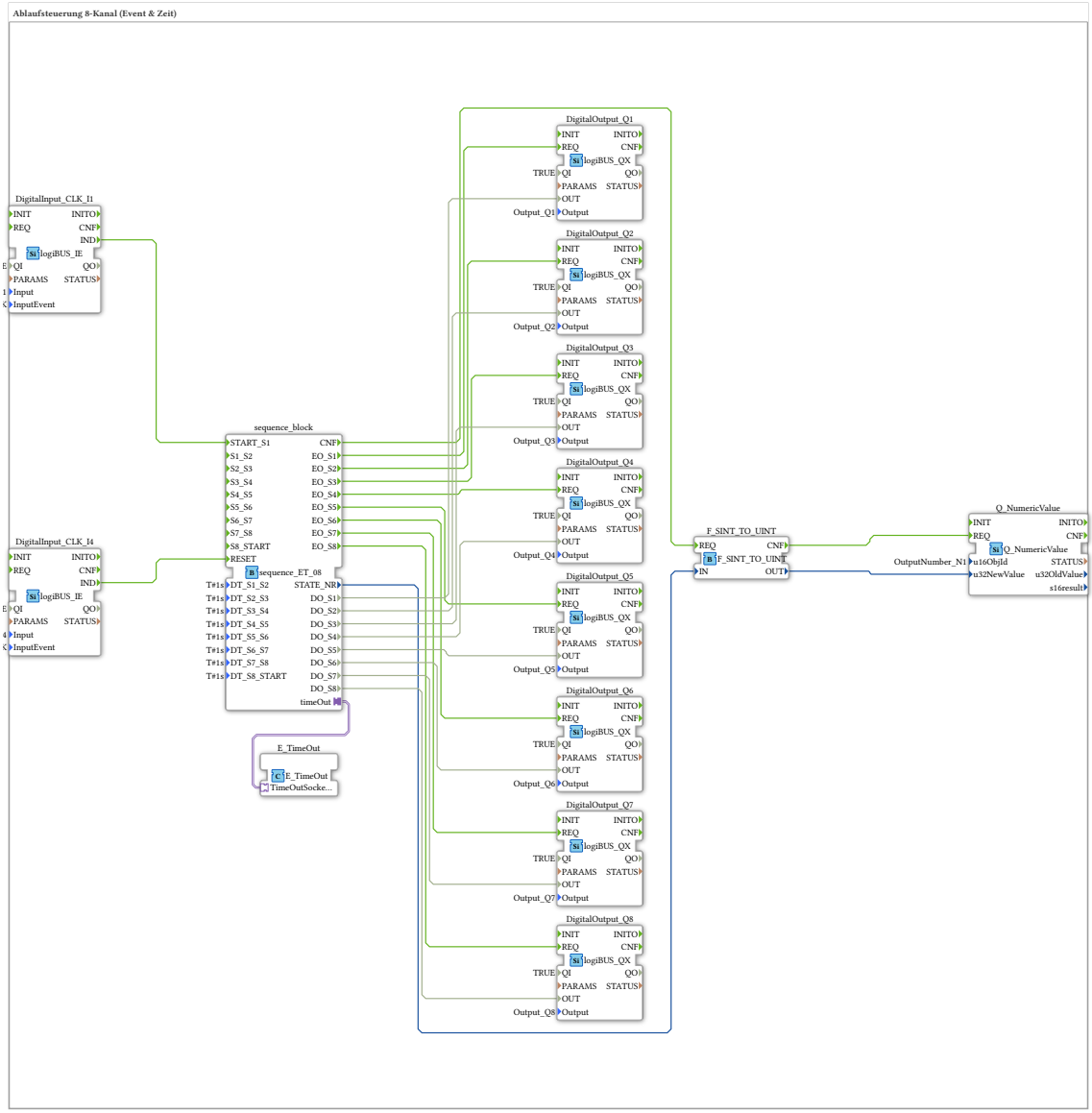
Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_045`. Wir bauen eine kombinierte Event- und Zeit-Schrittkette.

Ziel der Übung

Realisierung einer Abfolge von 8 Ausgängen (Q1, Q2, Q3, Q4, Q5, Q6, Q7, Q8), gestartet und zurückgesetzt über Ereignisse (Taster), wobei die einzelnen Schritte zusätzlich über feste Zeiten (`DT_Sx_Sy`, je `T#1s`) weitergeschaltet werden - eine Kombination aus Start/Reset per Taster und automatischem Zeittakt zwischen den Schritten.

Beschreibung und Komponenten

Die Subapplikation `Uebung_045` nutzt den kombinierten Sequenzer `sequence_ET_08`.



- **`sequence_ET_08`**: Startet auf Ereignis (`I1`), schaltet die 8 Schritte danach zeitgesteuert weiter.
- **`Q_NumericValue`**: Zeigt die aktuelle Schrittnummer auf dem ISOBUS-Terminal an.

Funktionsweise

1. Start durch Taster `I1` -> Sequenz springt zu `S1`.
2. Jeder Ausgang (`Q1, Q2, Q3, Q4, Q5, Q6, Q7, Q8`) wird für die eingestellte Zeit aktiv geschaltet, bevor automatisch zum nächsten Schritt gewechselt wird.
3. Nach dem letzten Schritt endet die Sequenz.
4. Reset durch Taster `I4` -> Alles aus.

2.2.1.249 Uebung_046: Ablaufsteuerung 8-Kanal (Event & Zeit) ENDLOS

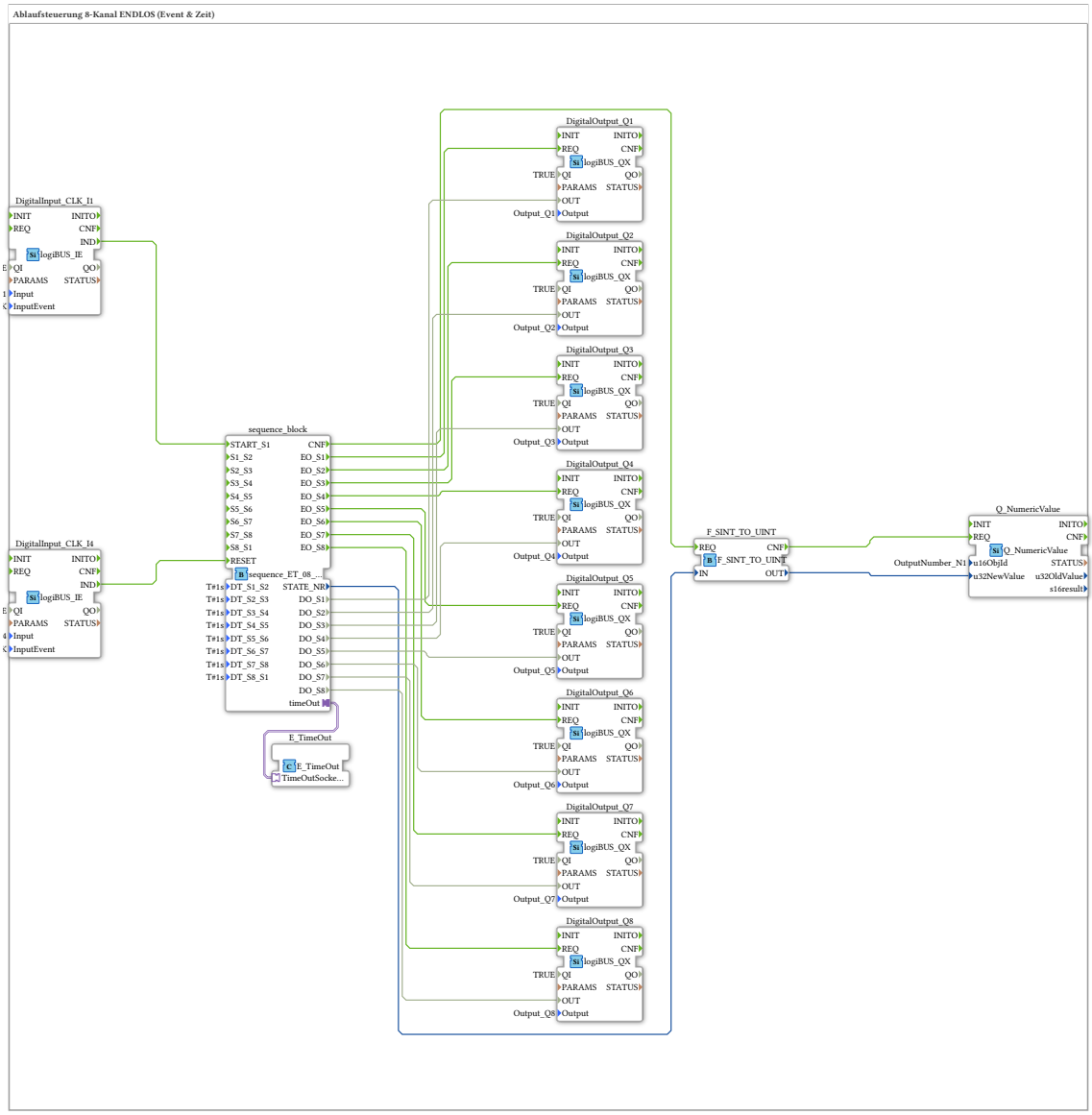
Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_046`. Wir bauen eine kombinierte Event- und Zeit-Schrittkette.

Ziel der Übung

Realisierung einer Abfolge von 8 Ausgängen (Q1, Q2, Q3, Q4, Q5, Q6, Q7, Q8), gestartet und zurückgesetzt über Ereignisse (Taster), wobei die einzelnen Schritte zusätzlich über feste Zeiten (`DT_Sx_Sy`, je `T#1s`) weitergeschaltet werden - eine Kombination aus Start/Reset per Taster und automatischem Zeittakt zwischen den Schritten.

Beschreibung und Komponenten

Die Subapplikation `Uebung_046` nutzt den kombinierten Sequenzer `sequence_ET_08_loop`.



- **`sequence_ET_08_loop`**: Startet auf Ereignis (`I1`), schaltet die 8 Schritte danach zeitgesteuert weiter.
- **`Q_NumericValue`**: Zeigt die aktuelle Schrittnummer auf dem ISOBUS-Terminal an.

Funktionsweise

1. Start durch Taster `I1` -> Sequenz springt zu `S1`.
2. Jeder Ausgang (`Q1, Q2, Q3, Q4, Q5, Q6, Q7, Q8`) wird für die eingestellte Zeit aktiv geschaltet, bevor automatisch zum nächsten Schritt gewechselt wird.
3. Nach dem letzten Schritt kehrt die Sequenz automatisch zu S1 zurück (ENDLOS).
4. Reset durch Taster `I4` -> Alles aus.

2.2.1.250 Uebung_047: Reine Event-Schrittkette 4-Kanal

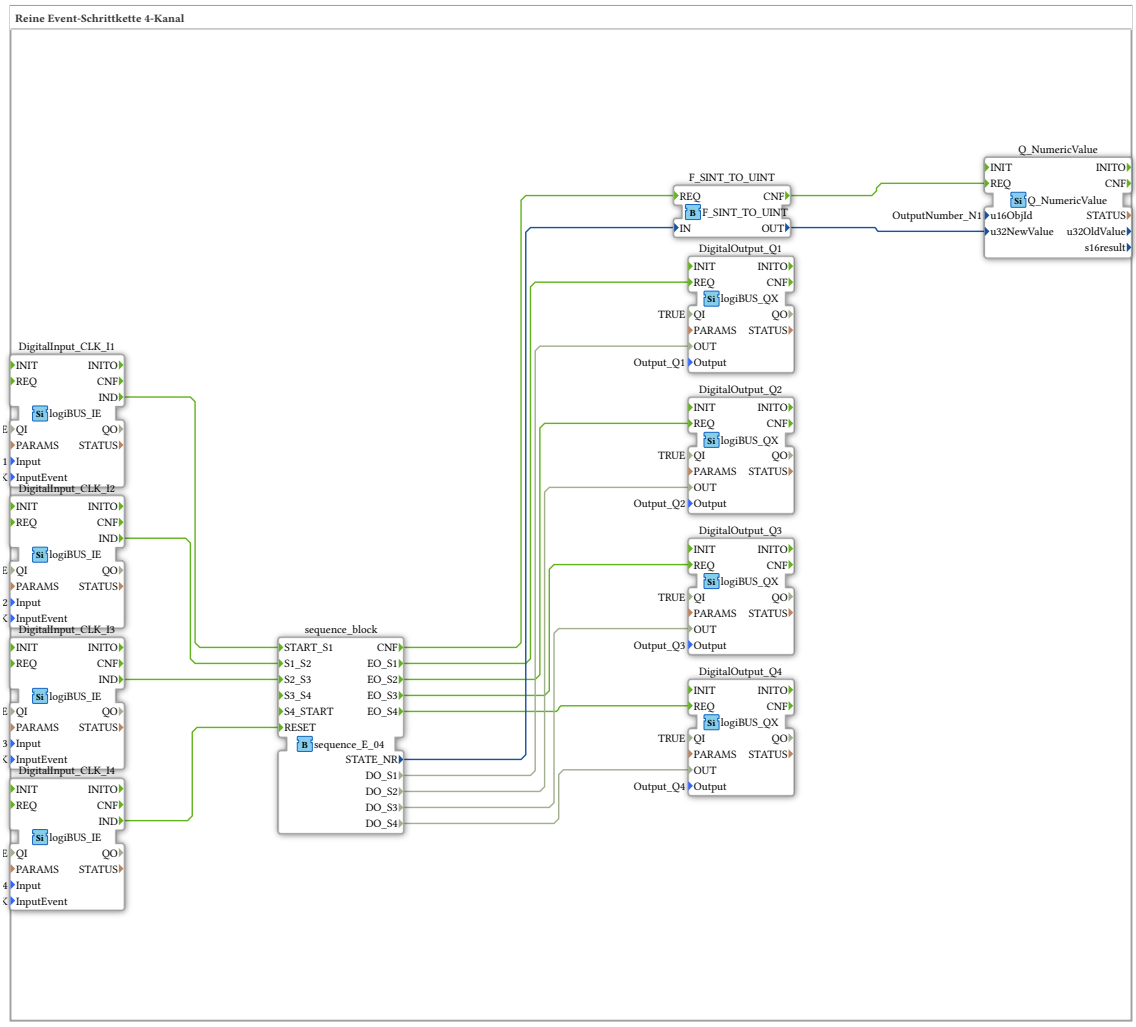
Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_047`. Wir bauen eine rein ereignisgesteuerte Schrittkette.

Ziel der Übung

Realisierung einer Abfolge von 4 Ausgängen (Q1, Q2, Q3, Q4), bei der jeder Schritt ausschließlich durch ein externes Ereignis (Tasterdruck) weitergeschaltet wird - keine automatische Zeitsteuerung.

Beschreibung und Komponenten

Die Subapplikation `Uebung_047` nutzt den Sequenzer `sequence_E_04`, der 4 Zustände rein ereignisgesteuert durchläuft.



Jeder Übergang zwischen den Schritten wird durch eine eigene Tasterbetätigung (`I1` bis `I4`) ausgelöst - die Sequenz wartet an jedem Schritt, bis der nächste Taster gedrückt wird.

- **`Q_NumericValue`**: Zeigt die aktuelle Schrittnummer auf dem ISOBUS-Terminal an.

Funktionsweise

1. Start durch Taster `I1` -> Sequenz springt zu `S1`, Ausgang `Q1` wird aktiv.
2. Jeder weitere Tasterdruck schaltet zum nächsten Schritt weiter.
3. Nach dem letzten Schritt bleibt die Sequenz stehen, bis sie zurückgesetzt wird.
4. Reset durch den letzten Taster -> Alles aus.

2.2.1.251 Uebung_048: Reine Event-Schrittkette 4-Kanal ENDLOS

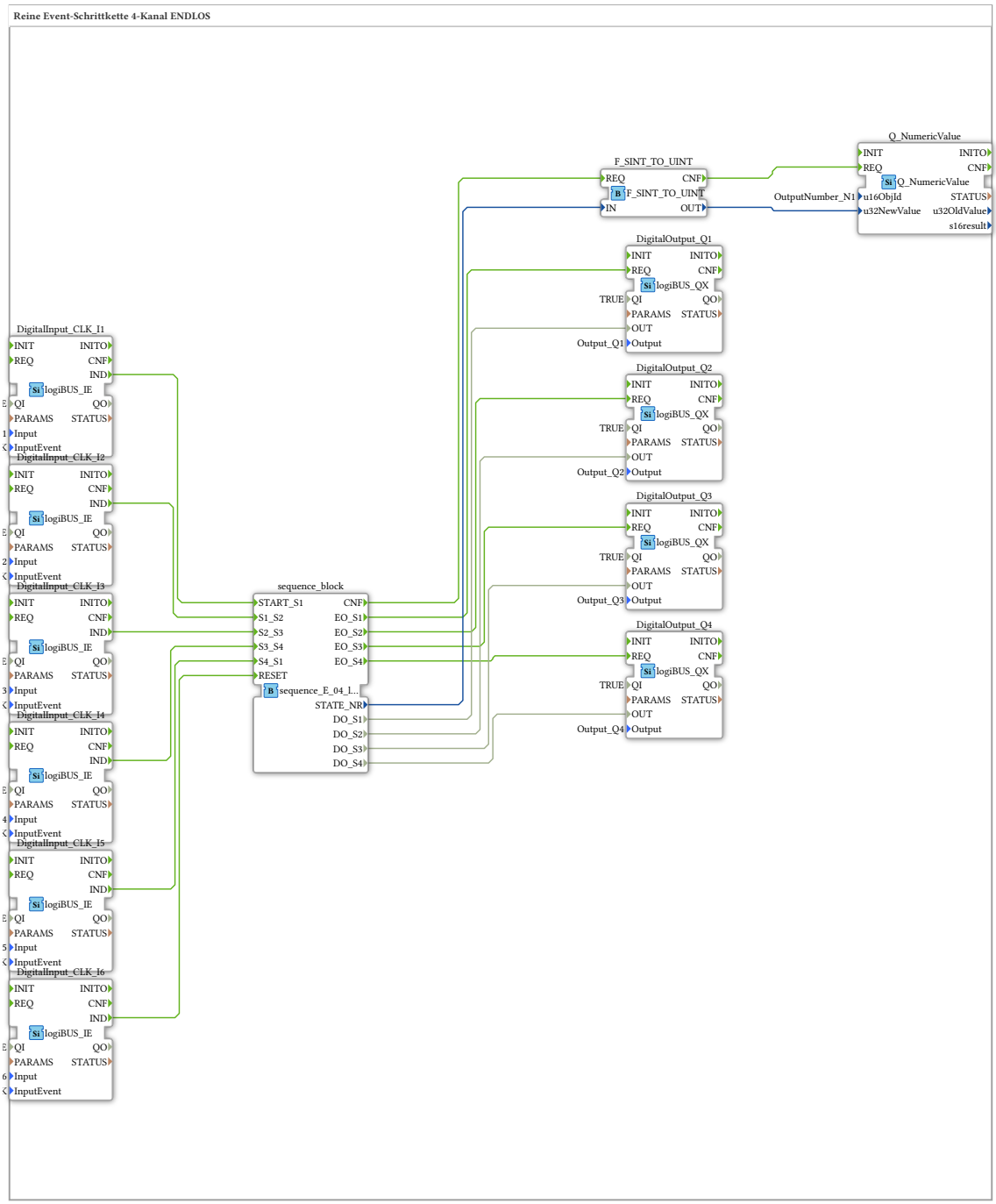
Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_048`. Wir bauen eine rein ereignisgesteuerte Schrittkette.

Ziel der Übung

Realisierung einer Abfolge von 4 Ausgängen (Q1, Q2, Q3, Q4), bei der jeder Schritt ausschließlich durch ein externes Ereignis (Tasterdruck) weitergeschaltet wird - keine automatische Zeitsteuerung.

Beschreibung und Komponenten

Die Subapplikation `Uebung_048` nutzt den Sequenzer `sequence_E_04_loop`, der 4 Zustände rein ereignisgesteuert durchläuft.



Jeder Übergang zwischen den Schritten wird durch eine eigene Tasterbetätigung (`I1` bis `I6`) ausgelöst - die Sequenz wartet an jedem Schritt, bis der nächste Taster gedrückt wird.

- **`Q_NumericValue`**: Zeigt die aktuelle Schrittnummer auf dem ISOBUS-Terminal an.

Funktionsweise

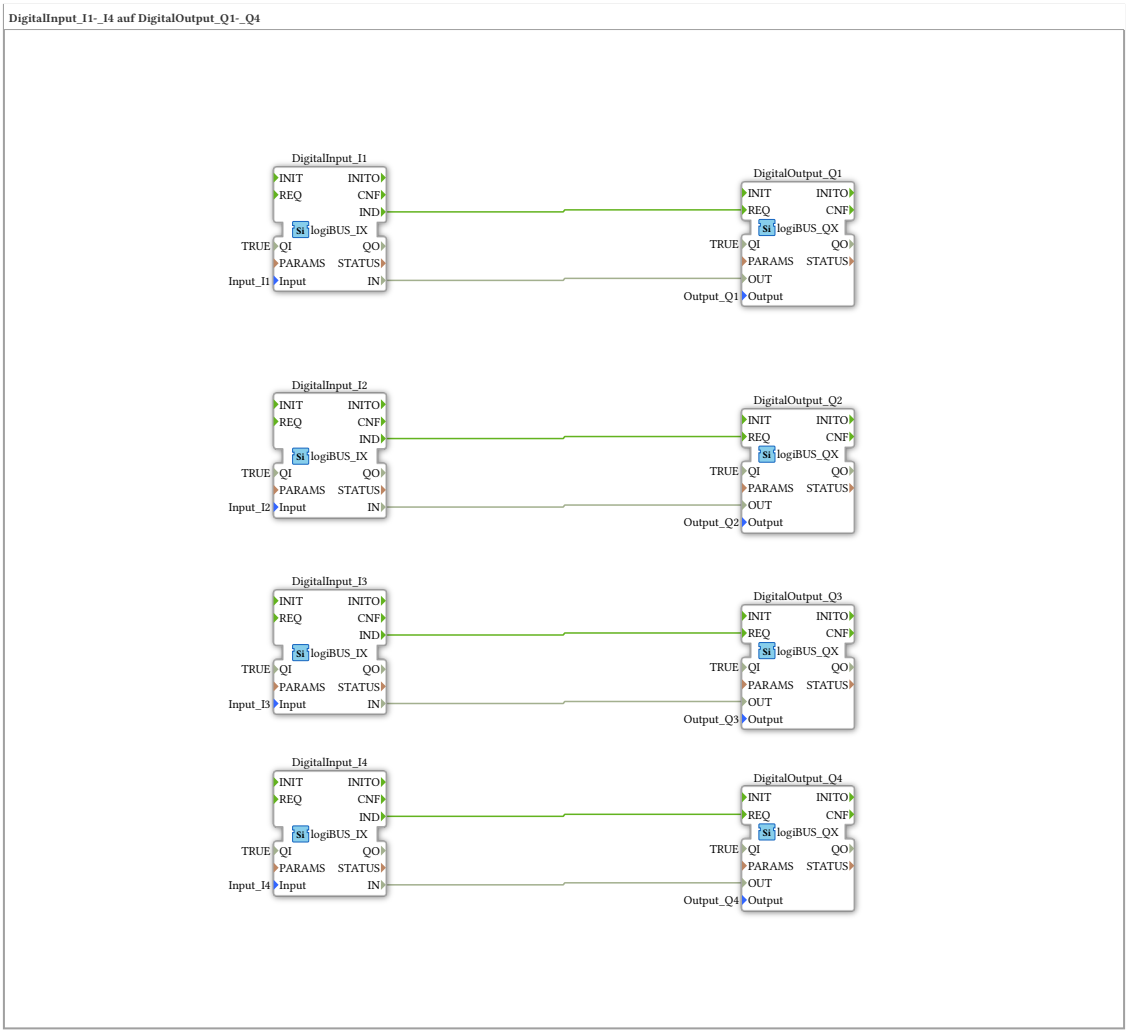
1. Start durch Taster `I1` -> Sequenz springt zu `S1`, Ausgang `Q1` wird aktiv.
2. Jeder weitere Tasterdruck schaltet zum nächsten Schritt weiter.
3. Nach dem letzten Schritt kehrt die Sequenz automatisch zu S1 zurück (ENDLOS).
4. Reset durch den letzten Taster -> Alles aus.

2.2.1.252 Uebung_049: DigitalInput_I1-_I4 auf DigitalOutput_Q1-_Q4

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_049`. Diese Übung dient der Übung von umfangreichen Punkt-zu-Punkt-Verbindungen.

🎧 Podcast



- 4diac IDE: Wie der IEC 61499 Standard die Industrieautomatisierung revolutioniert
- IEC 61499 vs. 61131: Brauchen wir einen neuen Standard für IIoT? Analyse einer hitzigen Debatte um Verteilte Intelligenz
- IEC 61499: Befreit der neue Standard die Industrieautomation? Ein Vergleich mit 61131 und die Brücke zwischen OT & IT.
- IEC 61499: Revolution der Industrieautomation – Warum der neue Standard Ihre Systeme fit für die Zukunft macht
- 4diac IDE: Dein Open-Source-Werkzeugkasten für verteilte Industrieautomatisierung nach IEC 61499

Übersicht

In `Uebung_049.SUB` werden vier digitale Eingänge (`I1` bis `I4`) direkt auf vier digitale Ausgänge (`Q1` bis `Q4`) gemappt. Dies ist die Basisform der Signalweiterleitung ohne Logik oder Strukturierung, bei der jeder Kanal über eigene Event- und Data-Connections verfügt. Es dient primär dem Training der manuellen Verdrahtung in der 4diac-IDE.

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

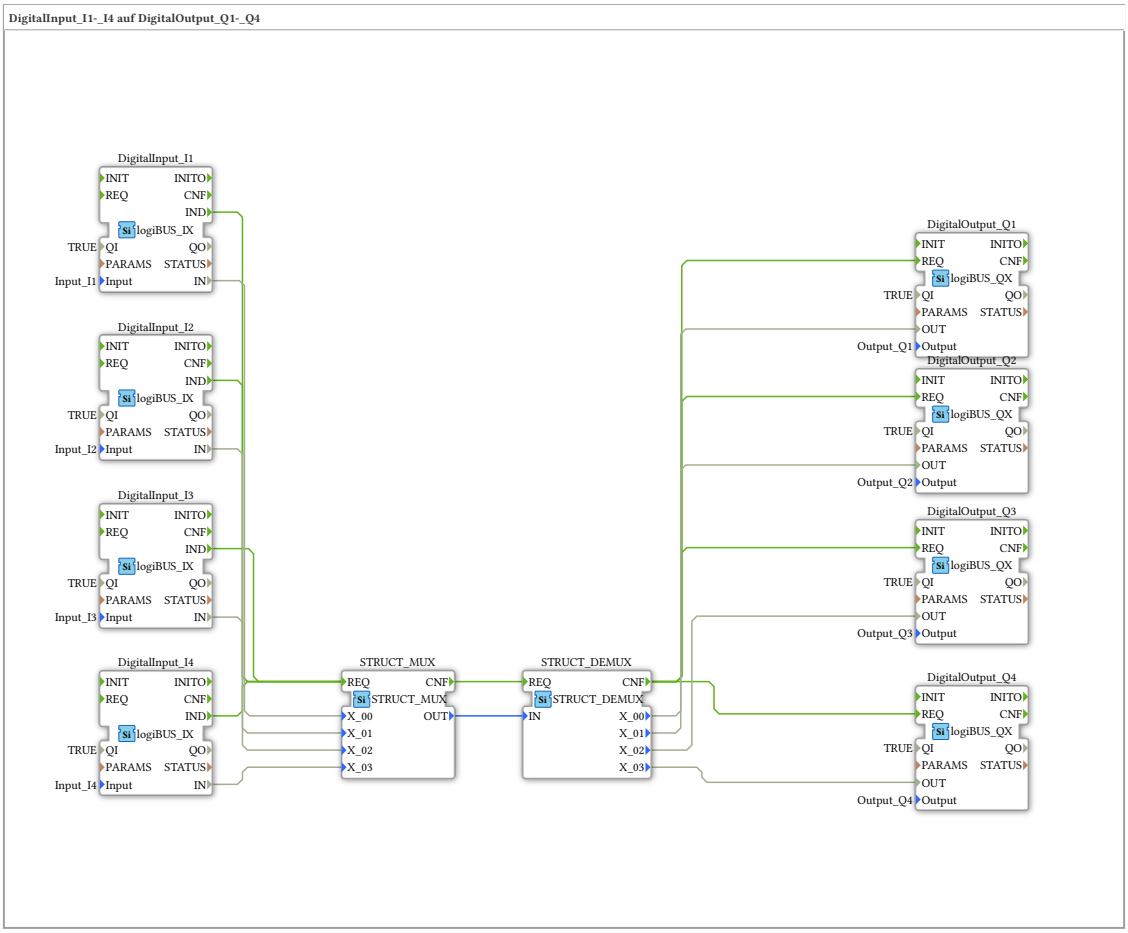
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.2.1.253 Uebung_051: DigitalInput_I1-_I4 auf DigitalOutput_Q1-_Q4



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_051`. Hier wird gezeigt, wie man viele Einzelsignale zu einem Paket (Struktur) zusammenfasst, um sie effizienter durch das Programm zu leiten.

🎧 Podcast



- Automatisierung entschlüsselt: Leiten, Steuern, Regeln – Die unsichtbare Sprache der Technik (DIN IEC 60050-351)
- Infineon CAN-Transceiver TLE9250V versus TLE9351VSJ
- Infineon TLE9351VSJ der unsichtbare Auto-Bodyguard
- Land- und Forstwirtschaft 4.0: Das Fundament der Sicherheit – Analyse der DIN EN ISO 25119-1 und der

---

Ziel der Übung

Verwendung von `STRUCT_MUX` und `STRUCT_DEMUX`. In großen Systemen ist es unübersichtlich, hunderte Einzelkabel zu ziehen. Stattdessen werden Signale gebündelt („gemultiplext“), über eine einzige Verbindung transportiert und am Zielort wieder entpackt.

Beschreibung und Komponenten

Die Subapplikation `Uebung_051.SUB` nutzt strukturierte Datentypen zur Signalübertragung.

Funktionsbausteine (FBs)

- **`STRUCT_MUX`**: Packt 4 einzelne Digitalsignale in einen strukturierten Datentyp (hier `ST04X`).
- **`STRUCT_DEMUX`**: Entnimmt der Struktur wieder die 4 Einzelsignale.

Funktionsweise

1. Die vier Taster liefern ihre Signale an die Eingänge `X_00` bis `X_03` des MUX.
2. Ein Klick auf einen beliebigen Taster triggert den `REQ` des MUX.
3. Der MUX erstellt ein Datenpaket (`OUT`), das alle 4 Zustände gleichzeitig enthält.
4. Über eine **einzige** Datenverbindung wandert dieses Paket zum DEMUX.
5. Der DEMUX zerlegt das Paket wieder und steuert die vier Lampen `Q1` bis `Q4` an.

Dies reduziert die Anzahl der Verbindungsleitungen im Hauptprogramm massiv und erhöht die Übersichtlichkeit.

Anwendungsbeispiel

Kabelbaum-Abstraktion: Stellen Sie sich vor, 16 Sensoren am Heck einer Maschine müssen zur Kabine geleitet werden. In der Software werden diese 16 Signale im Heck zu einer Struktur „Heck_Sensoren“ zusammengefasst. Nur diese eine Struktur wird durch die Programmlogik bis zur Kabinen-Ansicht gereicht, wo sie dann wieder in Einzelwerte für das Display zerlegt wird.

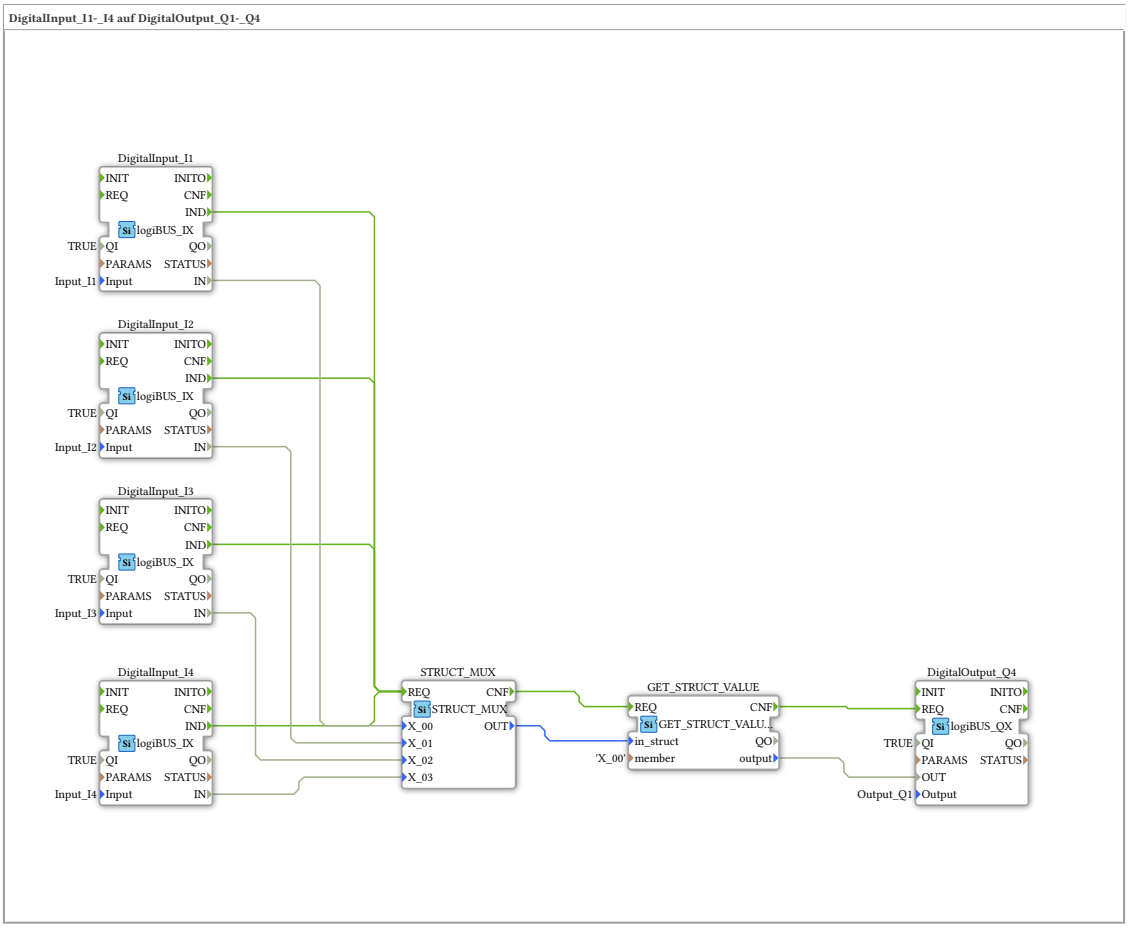
2.2.1.254 Uebung_052: DigitalInput_I1-_I4 auf DigitalOutput_Q1-_Q4



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_052`.

Übersicht

In dieser Variante wird gezeigt, wie man ein einzelnes Signal aus einer Struktur extrahiert, ohne alle Kanäle auspacken zu müssen. Über den Baustein `GET_STRUCT_VALUE` und den Parameter `member = 'X_00'` wird gezielt nur der erste Kanal aus dem Datenpaket der Übung `051` abgegriffen und auf den Ausgang `Q4` gelegt. Dies ist nützlich, wenn in einem Modul nur eine bestimmte Information aus einem großen Datenbündel benötigt wird.



2.2.1.255 Uebung_053: DigitalInput_I1-_I4 auf DigitalOutput_Q1-_Q4

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_053`.

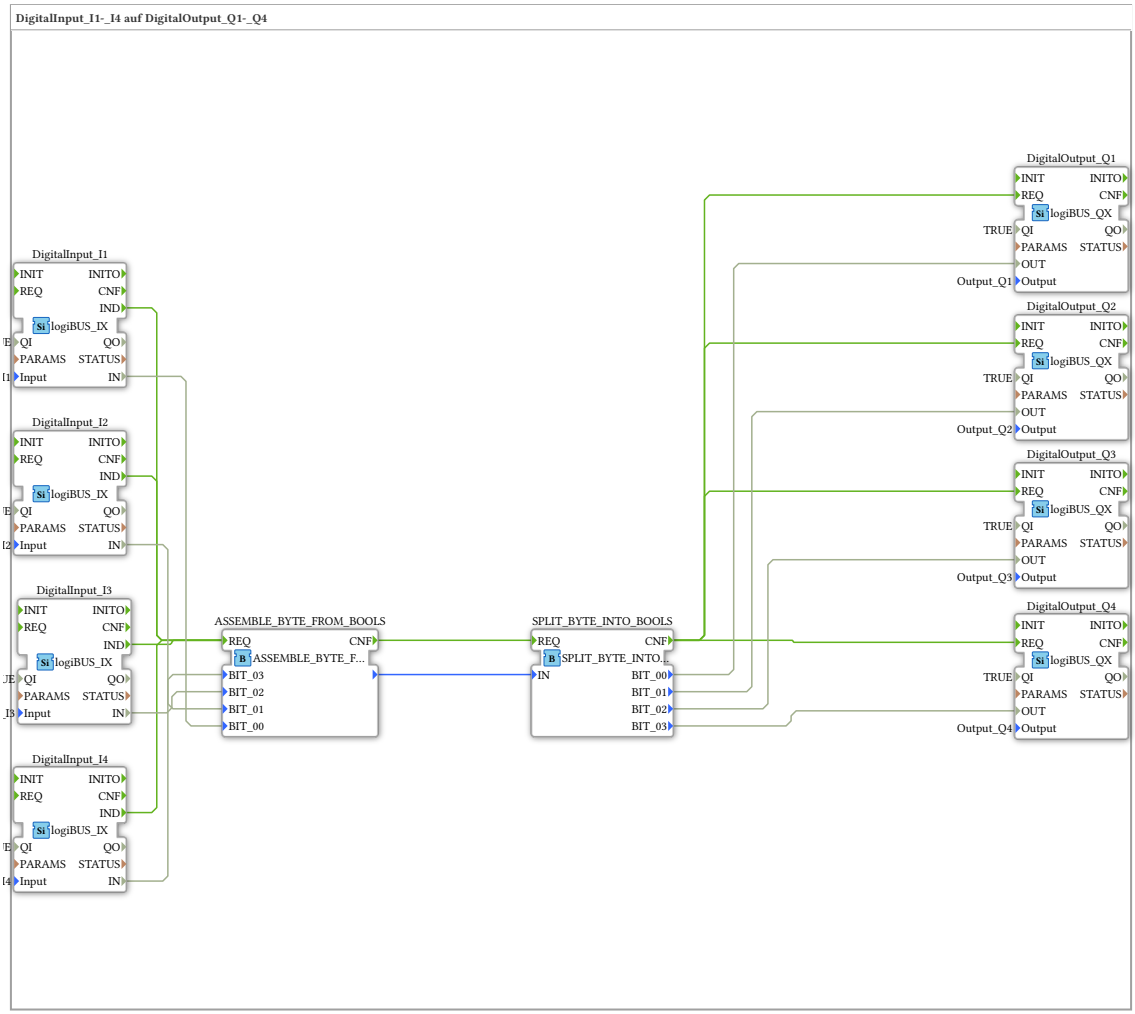
Ziel der Übung

Kombination von Bits zu einem Byte. Dies ist eine hardwarenahe Form der Bündelung, wie sie oft bei der Kommunikation mit Feldbus-Teilnehmern (z.B. CAN-Bus Nachrichten) vorkommt.

Beschreibung und Komponenten

Die Subapplikation `Uebung_053.SUB` nutzt Konvertierungs-Bausteine für den Datentyp `BYTE`.

Funktionsbausteine (FBs)



- **`ASSEMBLE_BYTE_FROM_BOOLS`**: Wandelt 8 Einzelbits (hier werden 4 genutzt) in einen 8-Bit Ganzzahlwert (BYTE) um.
- **`SPLIT_BYTE_INTO_BOOLS`**: Zerlegt das Byte wieder in seine einzelnen Bits.

Funktionsweise

Das Prinzip entspricht Übung 051, jedoch wird anstelle einer Software-Struktur ein standardisierter numerischer Datentyp (`BYTE`) als Container genutzt. Dies ist die effizienteste Form der Datenübertragung, da sie den Speicherverbrauch im Netzwerk minimiert.

2.2.1.256 Uebung_054: DigitalInput_I1-_I4 auf DigitalOutput_Q1-_Q4

NotebookLM

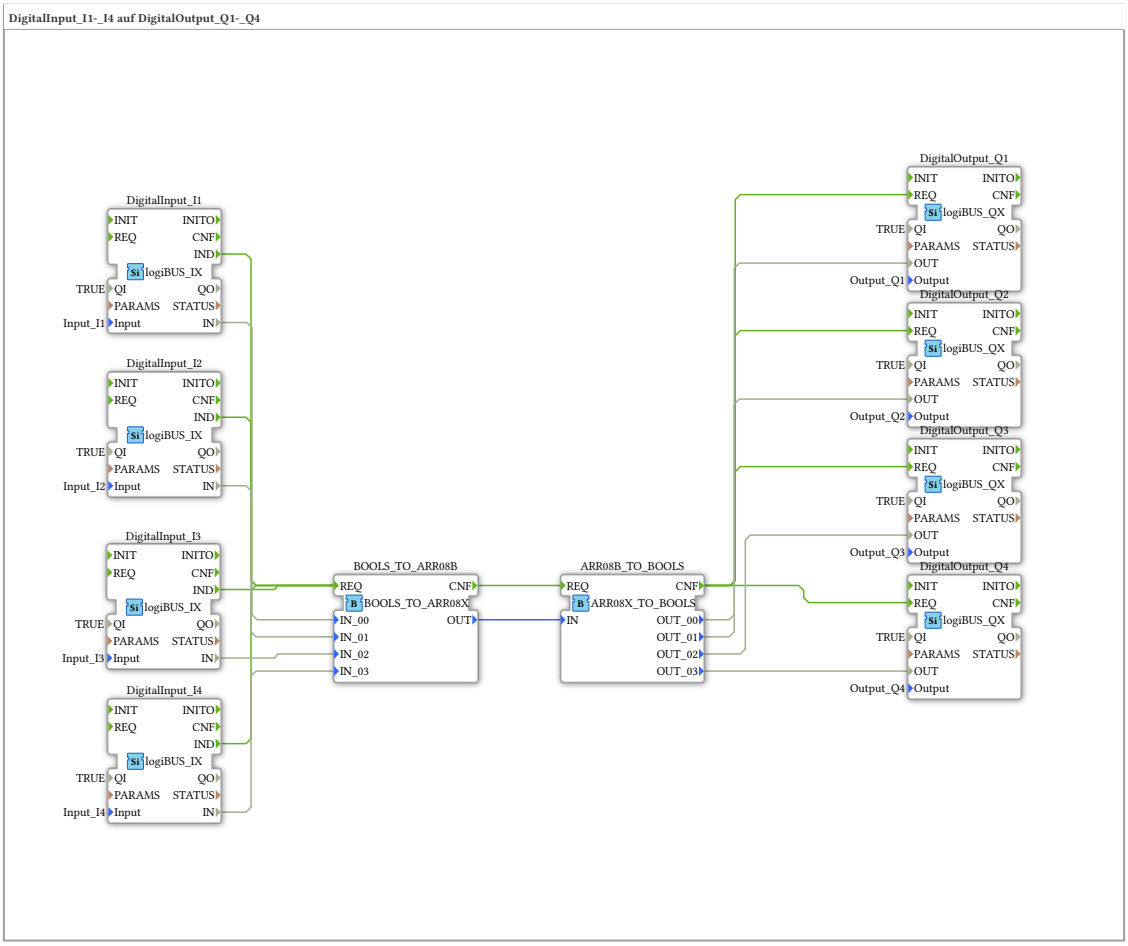
Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_054`. Dies ist die dritte Methode der Signalbündelung: Die Verwendung von Feldern (Arrays).

Ziel der Übung

Verwendung von `BOOLS_TO_ARR08X` und `ARR08X_TO_BOOLS`.

Beschreibung

In `Uebung_054.SUB` werden vier Digitalsignale in ein Array (eine indizierte Liste von Werten) verpackt. Im Gegensatz zur Struktur (wo jeder Kanal einen Namen hat, z.B. `X_00`) greift man bei einem Array über die Position (Index 0 bis 7) auf die Daten zu. Dies ist besonders vorteilhaft, wenn Signalpfade in Programmschleifen verarbeitet werden sollen.



2.2.1.257 Uebung_055: DigitalInput_I1 auf DigitalOutput_Q1

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_055`. Hier wird ein zentrales logiBUS-Konzept zur Übertragung von erweiterten Status-Informationen eingeführt: Das „Quarter“ (2-Bit Information).

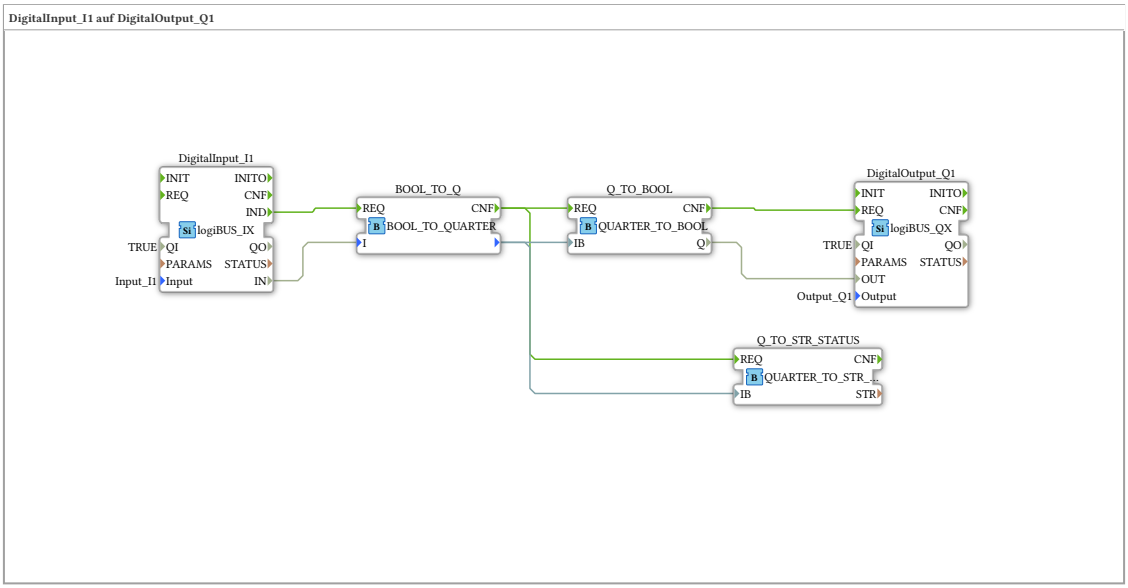
Ziel der Übung

Verständnis von erweiterten Signalzuständen. In professionellen Steuerungen reicht ein einfaches „An/Aus“ oft nicht aus. Man möchte auch wissen, ob ein Signal ungültig ist oder ein Fehler vorliegt. Ein „Quarter“ nutzt 2 Bit pro Kanal, um vier Zustände darzustellen (z.B. Aus, An, Fehler, Nicht Verfügbar).

Beschreibung und Komponenten

Die Subapplikation `Uebung_055.SUB` demonstriert die Wandlung zwischen einfachen booleschen Werten und logiBUS-Quartalen.

Funktionsbausteine (FBs)



- **`BOOL_TO_Q`**: Wandelt ein Standard-Bit in ein 2-Bit Quartal um.
- **`Q_TO_BOOL`**: Extrahiert das Haupt-Signal (An/Aus) wieder aus dem Quartal.
- **`QUARTER_TO_STR_STATUS`**: Wandelt den 2-Bit Code in einen lesbaren Text um (z.B. „STATUS_OFF“, „STATUS_ON“).

Funktionsweise

Das System reichert die Information an:

1. Der Taster `I1` liefert ein einfaches `TRUE/FALSE`.
2. `BOOL_TO_Q` macht daraus ein Quartal (z.B. FALSE ➡️ 00, TRUE ➡️ 01).
3. Dieses Paket (`QB`) kann nun durch das Programm geleitet werden.
4. Am Ende wird es wieder zerlegt: Die Lampe `Q1` erhält nur das An/Aus-Bit, während ein Diagnose-Baustein gleichzeitig den Textstatus („ON“) ermittelt.

Dies bildet die Grundlage für moderne Diagnose-Systeme in der Landtechnik.

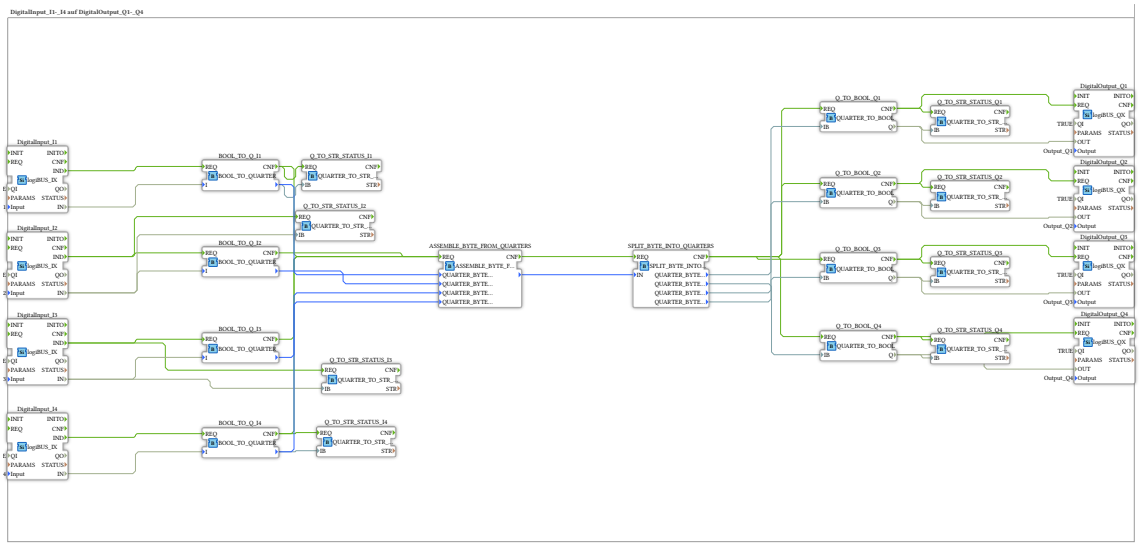
2.2.1.258 Uebung_056: DigitalInput_I1-_I4 auf DigitalOutput_Q1-_Q4

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_056`. Hier wird das Quarter-Konzept auf eine vierkanalige Struktur erweitert.

Übersicht

Die Subapplikation `Uebung_056.SUB` zeigt eine vollständige Diagnose-Pipeline:



1. **Eingabe**: Vier Taster (`I1`-`I4`) werden in Quartale gewandelt.
2. **Bündelung**: Vier Quartale (4 x 2 Bit = 8 Bit) werden über den Baustein `ASSEMBLE_BYTE_FROM_QUARTERS` zu einem einzigen Byte zusammengefasst.
3. **Transport**: Das Byte wird als Paket übertragen.
4. **Zerlegung**: `SPLIT_BYTE_INTO_QUARTERS` gewinnt die Information zurück.
5. **Ausgabe & Diagnose**: Die Signale steuern vier Lampen, während parallel für **jeden** Kanal ein Klartext-Status für das Terminal generiert wird.

Dies ist das Standard-Verfahren für die Übertragung von Sektions-Zuständen (z.B. bei einer Feldspritze) im logiBUS-System.

2.2.1.259 Uebung_057: Reine Event-Schrittkette 5-Kanal

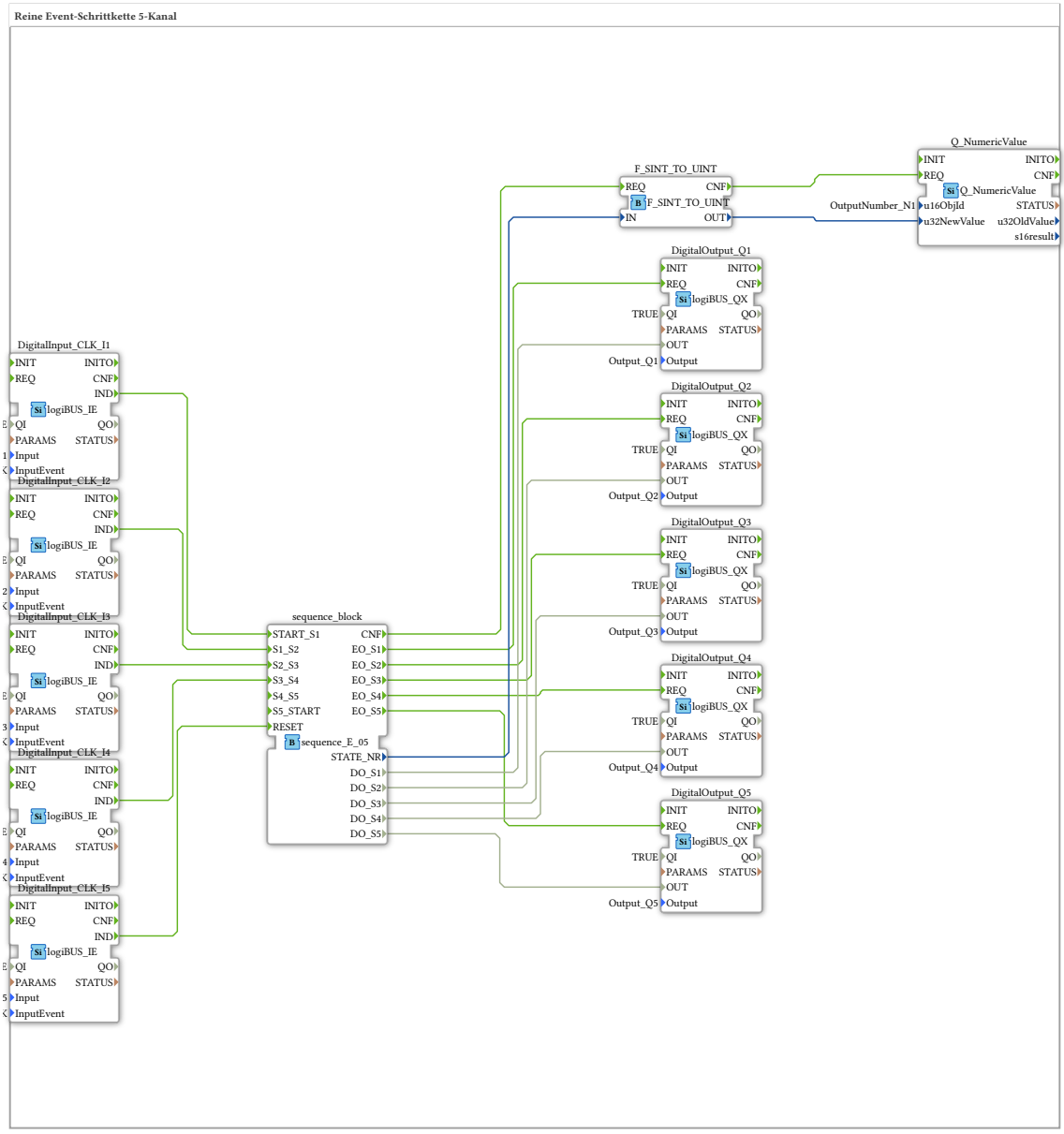
Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_057`. Wir bauen eine rein ereignisgesteuerte Schrittkette.

Ziel der Übung

Realisierung einer Abfolge von 5 Ausgängen (Q1, Q2, Q3, Q4, Q5), bei der jeder Schritt ausschließlich durch ein externes Ereignis (Tasterdruck) weitergeschaltet wird - keine automatische Zeitsteuerung.

Beschreibung und Komponenten

Die Subapplikation `Uebung_057` nutzt den Sequenzer `sequence_E_05`, der 5 Zustände rein ereignisgesteuert durchläuft.



Jeder Übergang zwischen den Schritten wird durch eine eigene Tasterbetätigung (`I1` bis `I5`) ausgelöst - die Sequenz wartet an jedem Schritt, bis der nächste Taster gedrückt wird.

- **`Q_NumericValue`**: Zeigt die aktuelle Schrittnummer auf dem ISOBUS-Terminal an.

Funktionsweise

1. Start durch Taster `I1` -> Sequenz springt zu `S1`, Ausgang `Q1` wird aktiv.
2. Jeder weitere Tasterdruck schaltet zum nächsten Schritt weiter.
3. Nach dem letzten Schritt bleibt die Sequenz stehen, bis sie zurückgesetzt wird.
4. Reset durch den letzten Taster -> Alles aus.

2.2.1.260 Uebung_058: Reine Event-Schrittkette 5-Kanal ENDLOS

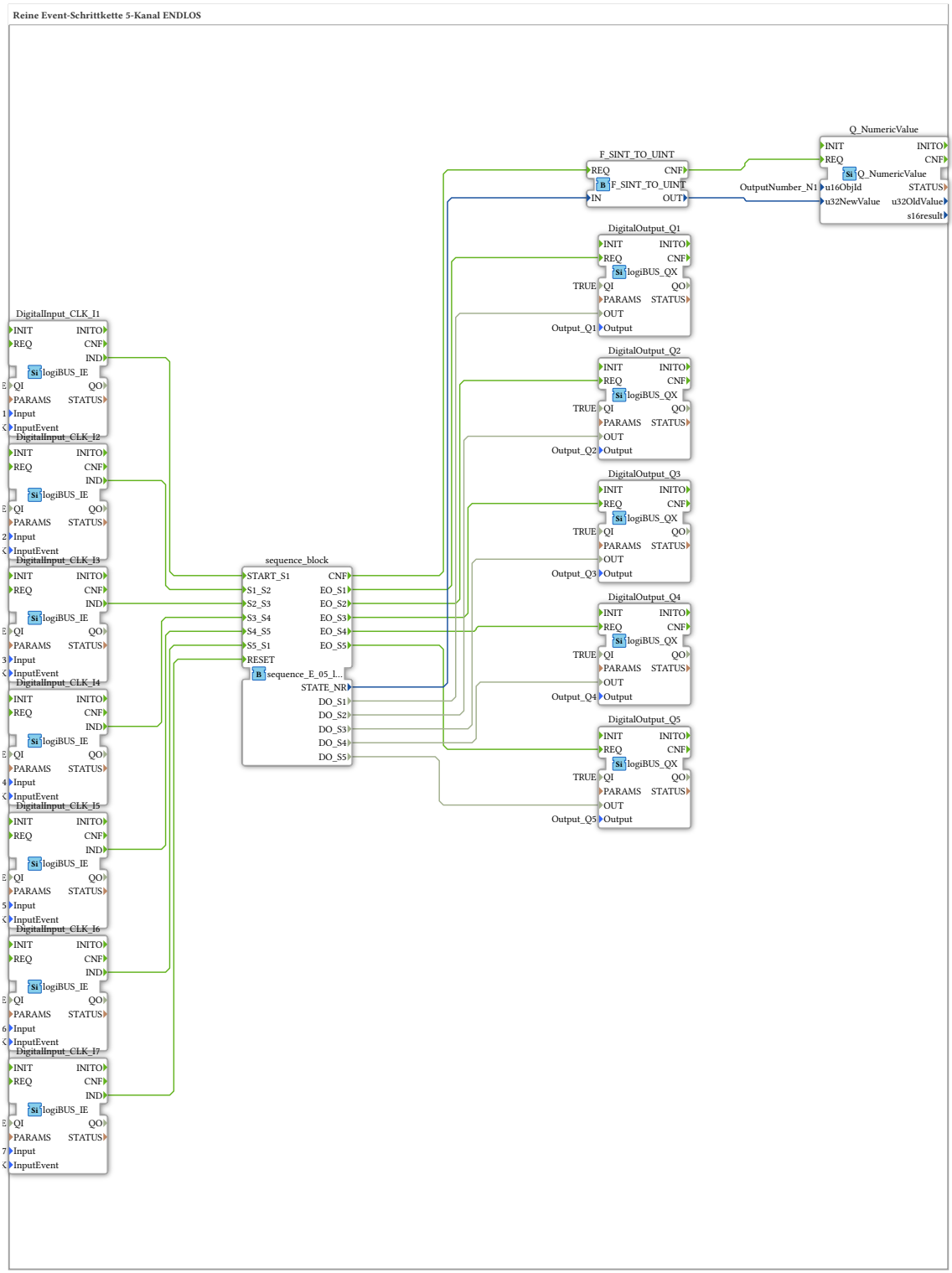
Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_058`. Wir bauen eine rein ereignisgesteuerte Schrittkette.

Ziel der Übung

Realisierung einer Abfolge von 5 Ausgängen (Q1, Q2, Q3, Q4, Q5), bei der jeder Schritt ausschließlich durch ein externes Ereignis (Tasterdruck) weitergeschaltet wird - keine automatische Zeitsteuerung.

Beschreibung und Komponenten

Die Subapplikation `Uebung_058` nutzt den Sequenzer `sequence_E_05_loop`, der 5 Zustände rein ereignisgesteuert durchläuft.



Jeder Übergang zwischen den Schritten wird durch eine eigene Tasterbetätigung (`I1` bis `I7`) ausgelöst - die Sequenz wartet an jedem Schritt, bis der nächste Taster gedrückt wird.

- **`Q_NumericValue`**: Zeigt die aktuelle Schrittnummer auf dem ISOBUS-Terminal an.

Funktionsweise

1. Start durch Taster `I1` -> Sequenz springt zu `S1`, Ausgang `Q1` wird aktiv.
2. Jeder weitere Tasterdruck schaltet zum nächsten Schritt weiter.
3. Nach dem letzten Schritt kehrt die Sequenz automatisch zu S1 zurück (ENDLOS).
4. Reset durch den letzten Taster -> Alles aus.

2.2.1.261 Uebung_059: Reine Event-Schrittkette 8-Kanal

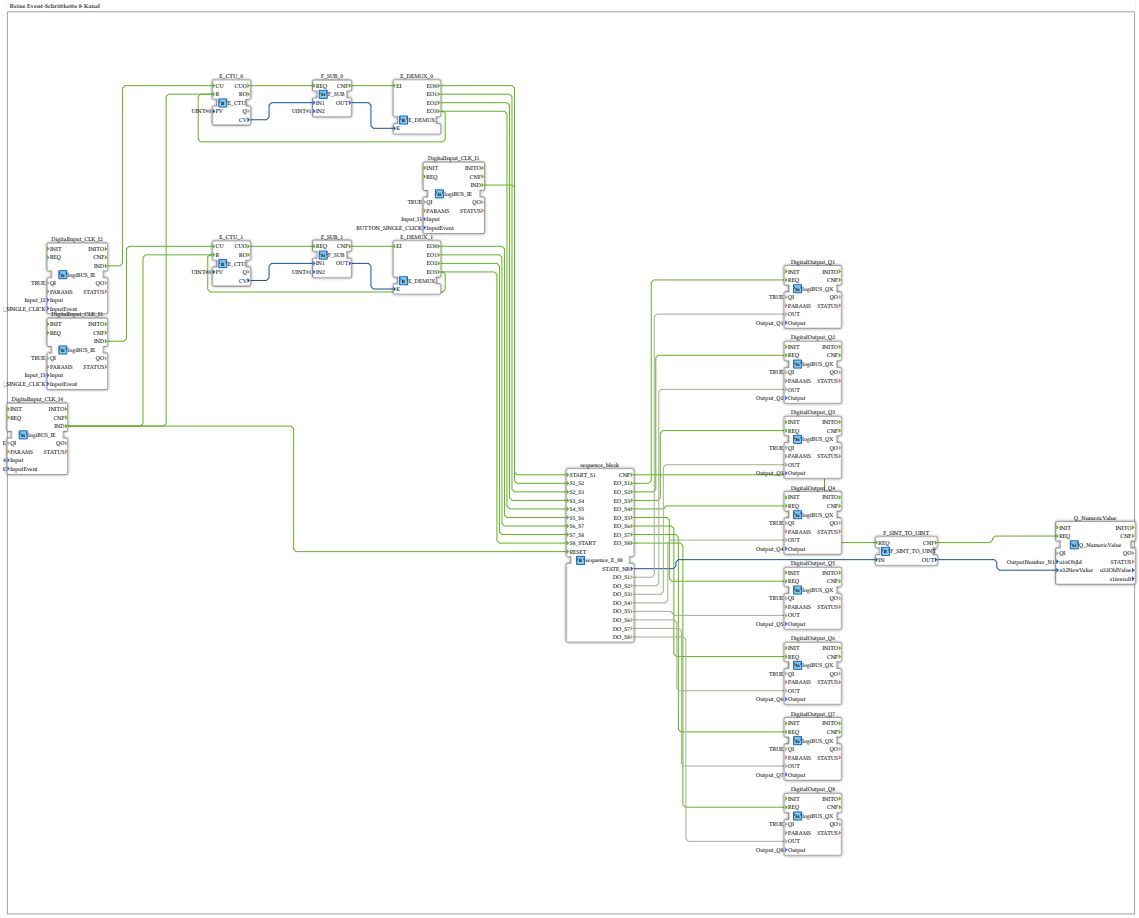
Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_059`. Wir bauen eine rein ereignisgesteuerte Schrittkette.

Ziel der Übung

Realisierung einer Abfolge von 8 Ausgängen (Q1, Q2, Q3, Q4, Q5, Q6, Q7, Q8), bei der jeder Schritt ausschließlich durch ein externes Ereignis (Tasterdruck) weitergeschaltet wird - keine automatische Zeitsteuerung.

Beschreibung und Komponenten

Die Subapplikation `Uebung_059` nutzt den Sequenzer `sequence_E_08`, der 8 Zustände rein ereignisgesteuert durchläuft.



Da nur die physischen Taster `I1-I4` zur Verfügung stehen, aber bis zu 8 Übergänge benötigt werden, zählt je ein `E_CTU`-Zähler die Betätigungen von `I2/I3` und ein nachgeschalteter `E_DEMUX` fächert das Zählergebnis auf die einzelnen `Sx_Sy`-Übergänge des Sequenzers auf - so lassen sich mit 2 Tastern 8 Zustandsübergänge auslösen, ohne für jeden Schritt einen eigenen Taster zu benötigen.

- **`Q_NumericValue`**: Zeigt die aktuelle Schrittnummer auf dem ISOBUS-Terminal an.

Funktionsweise

1. Start durch Taster `I1` -> Sequenz springt zu `S1`, Ausgang `Q1` wird aktiv.

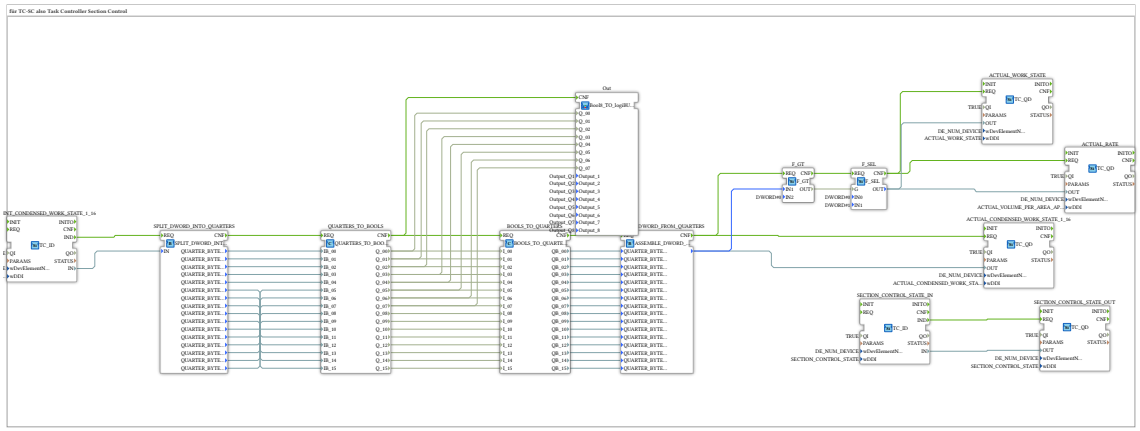
2. Jeder weitere Tasterdruck schaltet zum nächsten Schritt weiter.
3. Nach dem letzten Schritt bleibt die Sequenz stehen, bis sie zurückgesetzt wird.
4. Reset durch den letzten Taster -> Alles aus.

2.2.1.262 Uebung_060: für TC-SC also Task Controller Section Control



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_060`. Dies ist eine High-Level Übung für professionelle ISOBUS-Anwendungen im Bereich des Precision Farming.

Podcast



- Automatisierung entschlüsselt: Leiten, Steuern, Regeln – Die unsichtbare Sprache der Technik (DIN IEC 60050-351)

Ziel der Übung

Anbindung an einen ISOBUS Task Controller (TC). Es wird demonstriert, wie die automatische Teilbreitenschaltung (Section Control) und die Dokumentation von Arbeitszuständen und Ausbringraten realisiert werden.

Beschreibung und Komponenten

In `Uebung_060.SUB` werden Sollwerte vom Task Controller empfangen und Istwerte zurückgemeldet.

Funktionsbausteine (FBs)

- **`TC_ID`**: Empfängt Befehle vom Task Controller des Traktors (z.B. „Schalte Teilbreite 5 ein“).
- **`TC_QD`**: Meldet Daten an den Task Controller zurück (z.B. „Teilbreite 5 ist jetzt tatsächlich aktiv“).
- **Quarter-Logik**: Die Zustände der Teilbreiten werden als Quartale (2-Bit) übertragen, um auch Fehlerzustände (z.B. Kabelbruch am Ventil) an den TC melden zu können.
- **DDI (Data Dictionary Identifier)**: Spezifische Codes (z.B. `SETPOINT_CONDENSED_WORK_STATE`), die definieren, welche Information gerade übertragen wird.

Funktionsweise

1. **Sollwert-Empfang**: Der Task Controller sendet ein DWORD, das die Zustände von 16 Teilbreiten enthält.
2. **Verarbeitung**: Die Steuerung zerlegt dieses Wort in einzelne Quartale und dann in boolesche Signale für die Ventile (SubApp `Out`).
3. **Istwert-Rückmeldung**: Die tatsächlichen Zustände der Ventile werden wieder zu einem DWORD zusammengefügt und als „Actual Condensed Work State“ an den TC zurückgesendet.
4. **Arbeitsstatus**: Sobald mindestens eine Teilbreite aktiv ist (`F_GT`), meldet die Steuerung dem TC den „Actual Work State“ (Arbeit läuft), woraufhin der TC mit der Flächenaufzeichnung beginnt.

Anwendungsbeispiel

Automatische Teilbreitenschaltung bei einer Spritze: Der Traktor erkennt per GPS, dass ein Teil des Gestänges über eine bereits behandelte Fläche fährt. Der Task Controller sendet den Befehl „Teilbreite 1-4 AUS“. Das logiBUS-Programm empfängt diesen Befehl, schaltet die physischen Magnetventile ab und meldet dem Fahrer am Bildschirm den Erfolg durch den korrekten Ist-Status zurück.

2.2.1.263 Uebung_060_sub_Outputs

Hinweis

Diese Übung existiert nicht mehr als eigene Übung. Der Baustein wurde in die Bibliothek verschoben und heißt jetzt Bool8_TO_logiBUS_QX8. Beschreibung, Schnittstelle und Netzwerk stehen dort.

2.2.1.264 Uebung_061: Reine Zeitsteuerung 4-Kanal

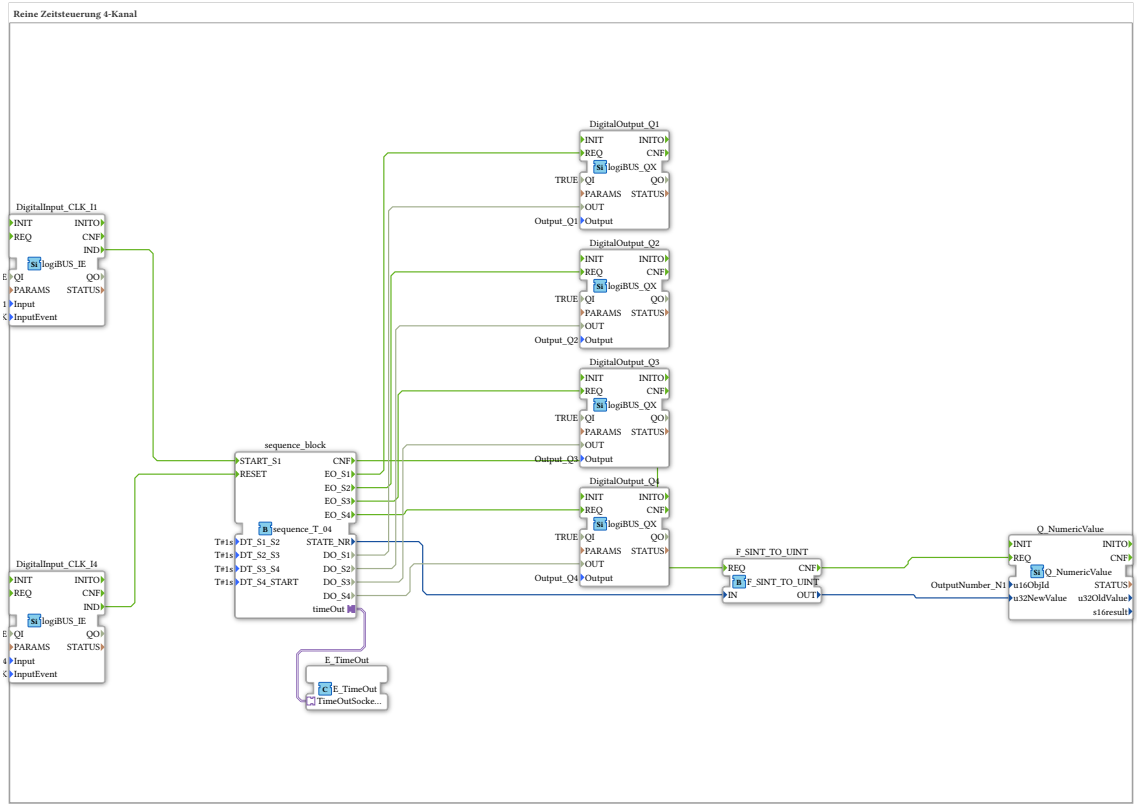
Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_061`. Wir bauen eine rein zeitgesteuerte Schrittkette.

Ziel der Übung

Realisierung einer automatischen Abfolge von 4 Ausgängen (Q1, Q2, Q3, Q4), bei der jeder Schritt ausschließlich nach Ablauf einer festen Wartezeit (`DT_S1_S2` usw., je `T#1s`) zum nächsten wechselt - ganz ohne weitere Tasterbetätigung während des Ablaufs.

Beschreibung und Komponenten

Die Subapplikation `Uebung_061` nutzt den Sequenzer `sequence_T_04`, der 4 Zustände (`S1` bis `S4`) rein zeitgesteuert durchläuft.



- **`sequence_T_04`**: Verwaltet die 4 Schritte, jeder Übergang erfolgt nach Ablauf der jeweiligen `DT_Sx_Sy`-Parameter.
- **`E_TimeOut`**: Überwacht die Sequenz als Watchdog.
- **`Q_NumericValue`**: Zeigt die aktuelle Schrittnummer auf dem ISOBUS-Terminal an.

Funktionsweise

1. Start durch Taster `I1` -> Sequenz springt zu `S1`.
2. Jeder Ausgang (`Q1, Q2, Q3, Q4`) wird für die eingestellte Zeit aktiv geschaltet, bevor automatisch zum nächsten Schritt gewechselt wird.
3. Nach dem letzten Schritt endet die Sequenz.
4. Reset durch Taster `I4` -> Alles aus, Sequenz zurück in den Grundzustand.

2.2.1.265 Uebung_062: Reine Zeitsteuerung 5-Kanal

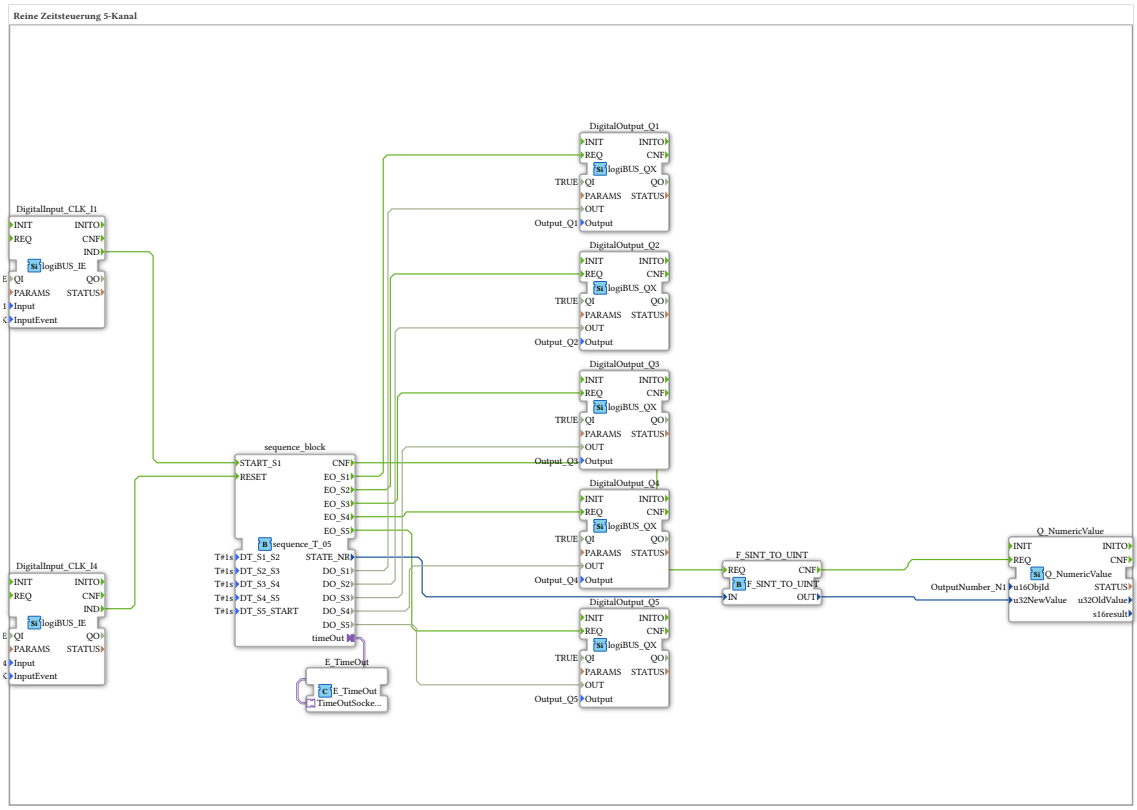
Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_062`. Wir bauen eine rein zeitgesteuerte Schrittkette.

Ziel der Übung

Realisierung einer automatischen Abfolge von 5 Ausgängen (Q1, Q2, Q3, Q4, Q5), bei der jeder Schritt ausschließlich nach Ablauf einer festen Wartezeit (`DT_S1_S2` usw., je `T#1s`) zum nächsten wechselt - ganz ohne weitere Tasterbetätigung während des Ablaufs.

Beschreibung und Komponenten

Die Subapplikation `Uebung_062` nutzt den Sequenzer `sequence_T_05`, der 5 Zustände (`S1` bis `S5`) rein zeitgesteuert durchläuft.



- **`sequence_T_05`**: Verwaltet die 5 Schritte, jeder Übergang erfolgt nach Ablauf der jeweiligen `DT_Sx_Sy`-Parameter.
- **`E_TimeOut`**: Überwacht die Sequenz als Watchdog.
- **`Q_NumericValue`**: Zeigt die aktuelle Schrittnummer auf dem ISOBUS-Terminal an.

Funktionsweise

1. Start durch Taster `I1` -> Sequenz springt zu `S1`.
2. Jeder Ausgang (`Q1, Q2, Q3, Q4, Q5`) wird für die eingestellte Zeit aktiv geschaltet, bevor automatisch zum nächsten Schritt gewechselt wird.
3. Nach dem letzten Schritt endet die Sequenz.
4. Reset durch Taster `I4` -> Alles aus, Sequenz zurück in den Grundzustand.

2.2.1.266 Uebung_063: Reine Zeitsteuerung 8-Kanal

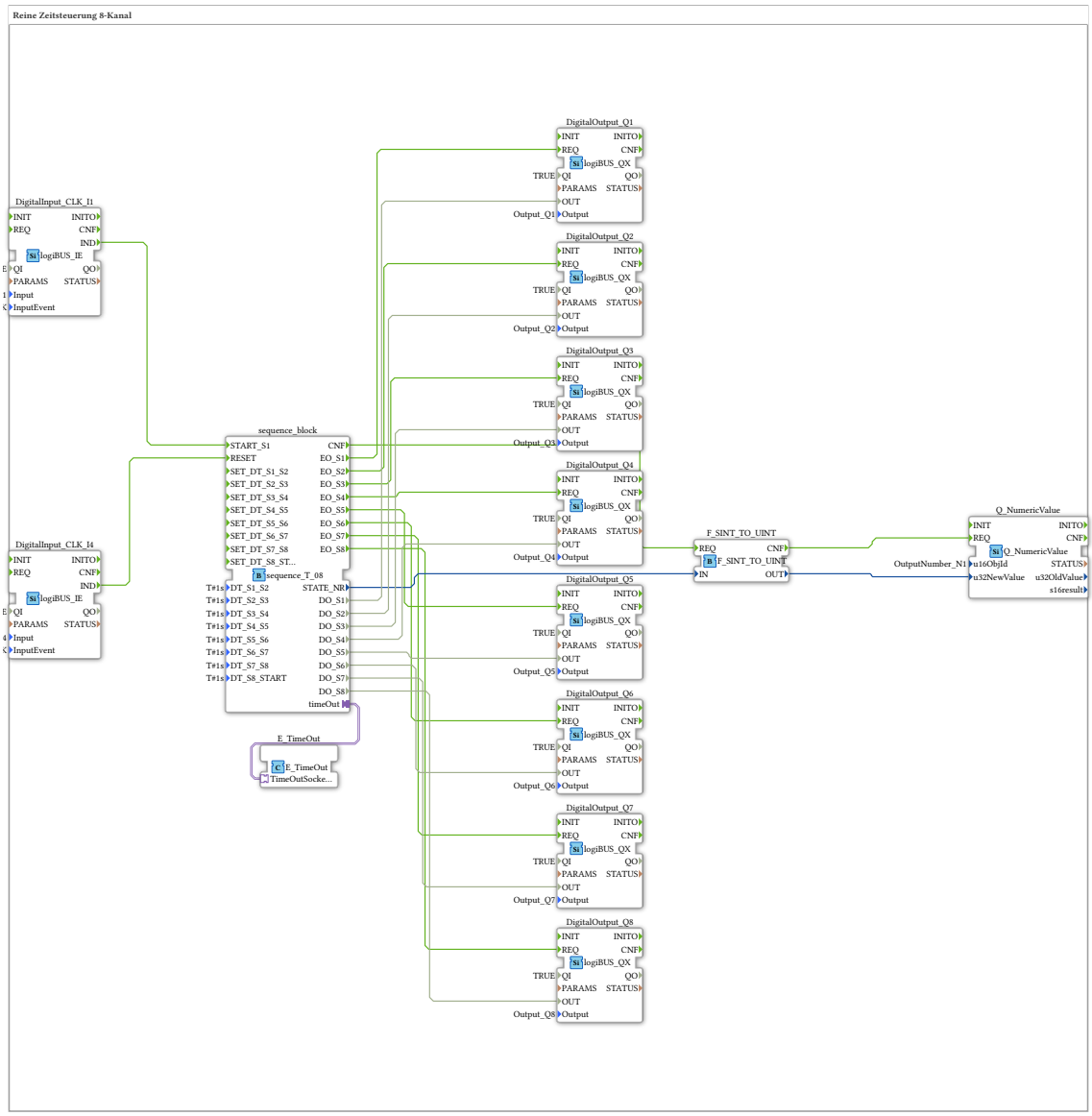
Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_063`. Wir bauen eine rein zeitgesteuerte Schrittkette.

Ziel der Übung

Realisierung einer automatischen Abfolge von 8 Ausgängen (Q1, Q2, Q3, Q4, Q5, Q6, Q7, Q8), bei der jeder Schritt ausschließlich nach Ablauf einer festen Wartezeit (`DT_S1_S2` usw., je `T#1s`) zum nächsten wechselt - ganz ohne weitere Tasterbetätigung während des Ablaufs.

Beschreibung und Komponenten

Die Subapplikation `Uebung_063` nutzt den Sequenzer `sequence_T_08`, der 8 Zustände (`S1` bis `S8`) rein zeitgesteuert durchläuft.



- **`sequence_T_08`**: Verwaltet die 8 Schritte, jeder Übergang erfolgt nach Ablauf der jeweiligen `DT_Sx_Sy`-Parameter.
- **`E_TimeOut`**: Überwacht die Sequenz als Watchdog.
- **`Q_NumericValue`**: Zeigt die aktuelle Schrittnummer auf dem ISOBUS-Terminal an.

Funktionsweise

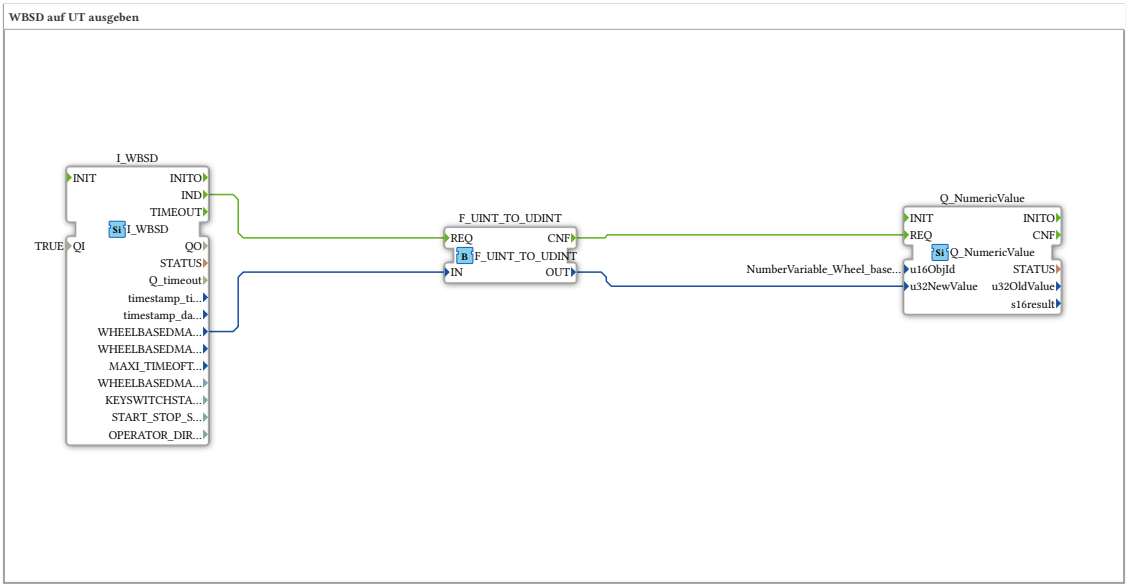
1. Start durch Taster `I1` -> Sequenz springt zu `S1`.

2. Jeder Ausgang (`Q1, Q2, Q3, Q4, Q5, Q6, Q7, Q8`) wird für die eingestellte Zeit aktiv geschaltet, bevor automatisch zum nächsten Schritt gewechselt wird.
3. Nach dem letzten Schritt endet die Sequenz.
4. Reset durch Taster `I4` -> Alles aus, Sequenz zurück in den Grundzustand.

2.2.1.267 Uebung_070: WBSD auf UT ausgeben

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_070`. Hier wird gezeigt, wie man Daten von der Traktor-ECU (TECU) einliest und auf dem Terminal visualisiert.

Podcast



- Der BTS7030-2EPA intelligenter Auto Stromwächter
- Der Intelligente Leistungsschalter: Wie der Infineon BTS7030 Relais und Sicherungen im Auto ersetzt
- Infineon BTS7030-2EPA: Intelligenter High-Side Leistungsschalter
- JBC Lötspitzen C470 vs. C245 vs. C210 vs. C115: Welche Spitze ist der Allrounder und wann brauchst du den Nano-Spezialisten?
- Verpolungsschutz in der Elektronik: Warum die ideale Diode (LM74700) MOSFETs und Schottky-Dioden in Effizienz und Kosten schlägt

---

Ziel der Übung

Verwendung des Bausteins `I_WBSD` (Wheel Based Speed and Distance). Ziel ist es, die vom Getriebe oder den Rädern des Traktors gemeldete Geschwindigkeit abzugreifen und als numerischen Wert an ein ISOBUS-Terminal zu senden.

Beschreibung und Komponenten

Die Subapplikation `Uebung_070.SUB` liest die ISOBUS-Nachricht WBSD ein und leitet sie an eine numerische Anzeige weiter.

Funktionsbausteine (FBs)

- **`I_WBSD`**: Typ `isobus::tecu::I_WBSD`. Dieser Baustein lauscht auf dem CAN-Bus nach den standardisierten TECU-Nachrichten für radbasierte Geschwindigkeit und Wegstrecke.
- **`Q_NumericValue`**: Sendet den Wert an das Objekt `NumberVariable_Wheel_based_machine_speed` im Terminal-Pool.

Funktionsweise

Die TECU sendet die Geschwindigkeitsdaten in festen Zeitintervallen (zyklisch) auf den ISOBUS.

1. Der Baustein `I_WBSD` empfängt eine neue Nachricht.
2. Er aktualisiert den Ausgang `WHEELBASEDMACHINESPEED` und feuert ein `IND`-Event.
3. Das Ereignis triggert die Anzeige am Terminal.
4. Der Fahrer sieht die aktuelle Geschwindigkeit des Traktors in Echtzeit auf seinem Display.

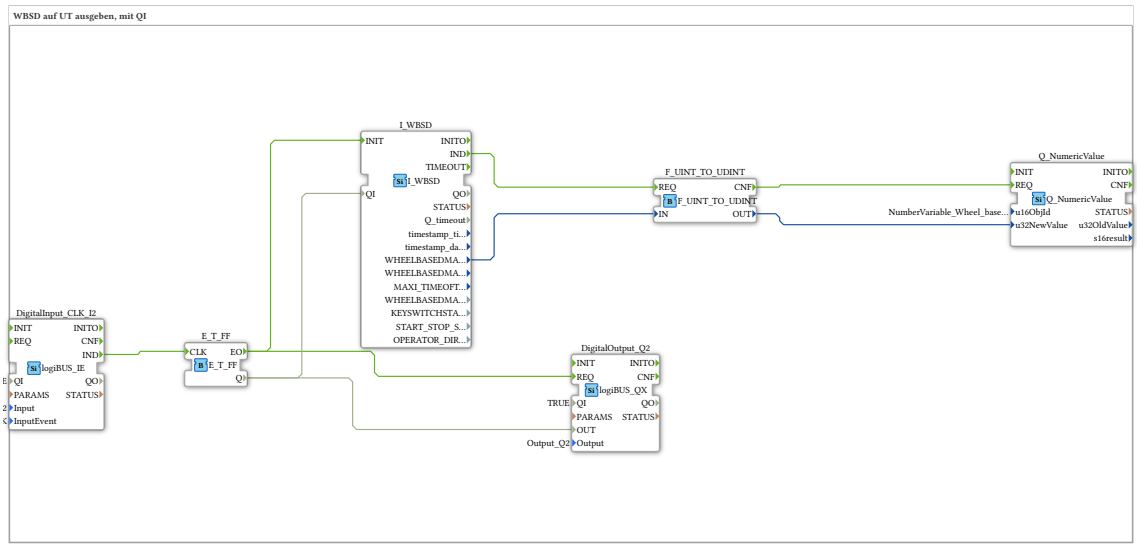
Anwendungsbeispiel

Überwachung der Fahrgeschwindigkeit: Bei der Ausbringung von Gülle oder Dünger ist die exakte Einhaltung der Geschwindigkeit entscheidend für die Dosierung. Die Anzeige am Terminal dient dem Fahrer als Kontrolle, ob er im optimalen Bereich fährt.

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

- 🌐 Interaktiver JBC-Lötspitzen Guide & Infografik auf ms-muc-docs.de
- 🌐 Diode & Halbleitergrundlagen auf ms-muc-docs.de

2.2.1.268 Uebung_070b: WBSD auf UT ausgeben, mit QI



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert die Ausgabe der radbasierten Maschinengeschwindigkeit (Wheel-based Machine Speed, WBSD) auf ein ISOBUS-Universal Terminal (UT). Dabei wird der Qualifier-Eingang **QI** des WBSD-Bausteins über einen T-Flipflop gesteuert, der mit einem Digitaleingang (Taster I2) getoggelt wird. Die Ausgabe des aktuellen Geschwindigkeitswerts erfolgt über den UT-Baustein **Q_NumericValue**.

Verwendete Funktionsbausteine (FBs)

- **I_WBSD**
  - Typ: `isobus::tecu::I_WBSD`
  - Stellt die radbasierte Maschinengeschwindigkeit als 16-Bit-Wert bereit.
  - Ereignisse: `IND` (Ausgang), `INIT` (Eingang)
  - Daten: `WHEELBASEDMACHINESPEED` (Ausgang), `QI` (Eingang)
- **Q_NumericValue**
  - Typ: `isobus::UT::Q::Q_NumericValue`
  - Zeigt einen numerischen Wert auf dem UT an.
  - Parameter: `u16ObjId` = `NumberVariable_Wheel_based_machine_speed`
  - Ereignisse: `REQ` (Eingang)
  - Daten: `u32NewValue` (Eingang)

- **F_UINT_TO_UDINT**
  - Typ: `iec61131::conversion::F_UINT_TO_UDINT`
  - Konvertiert einen 16-Bit-Wert in einen 32-Bit-Wert.
  - Ereignisse: `REQ` (Eingang), `CNF` (Ausgang)
  - Daten: `IN` (Eingang), `OUT` (Ausgang)
- **E_T_FF**
  - Typ: `iec61499::events::E_T_FF`
  - T-Flipflop: Bei jeder steigenden Flanke am Takteingang wird der Ausgangszustand umgeschaltet.
  - Ereignisse: `CLK` (Eingang), `EO` (Ausgang)
  - Daten: `Q` (Ausgang)
- **DigitalInput_CLK_I2**
  - Typ: `logiBUS::io::DI::logiBUS_IE`
  - Digitaler Eingang (I2) mit Ereignisauslösung bei Tastendruck (BUTTON_SINGLE_CLICK).
  - Parameter: `QI` = `TRUE`, `Input` = `Input_I2`, `InputEvent` = `BUTTON_SINGLE_CLICK`
  - Ereignisse: `IND` (Ausgang)
- **DigitalOutput_Q2**
  - Typ: `logiBUS::io::DQ::logiBUS_QX`
  - Digitaler Ausgang (Q2).
  - Parameter: `QI` = `TRUE`, `Output` = `Output_Q2`
  - Ereignisse: `REQ` (Eingang)
  - Daten: `OUT` (Eingang)

Programmablauf und Verbindungen

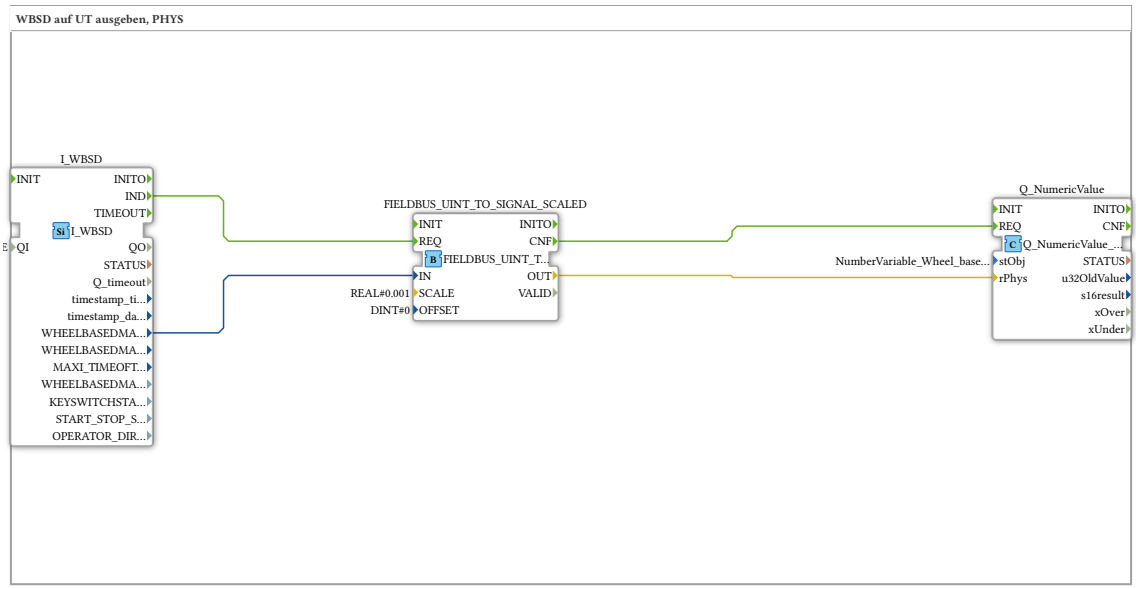
1. **Eingangssignal**: Der digitale Eingang **I2** wird als Taster genutzt. Bei einem einfachen Klick (Single Click) erzeugt der Baustein `DigitalInput_CLK_I2` ein Ereignis `IND`.
2. **T-Flipflop**: Dieses Ereignis gelangt an den Takteingang `CLK` des **E_T_FF**. Bei jedem Tastendruck wechselt der Ausgang `Q` seinen Zustand (Toggle).
3. **Steuerung von WBSD und Ausgang Q2**:
   - Der Flipflop-Ausgang `Q` wird auf den Qualifier-Eingang `QI` von **I_WBSD** sowie auf den Daten-Eingang `OUT` des Digitalausgangs **DigitalOutput_Q2** geschaltet.
   - Wenn der Flipflop den Zustand wechselt, wird gleichzeitig das Ereignis `EO` ausgelöst, das sowohl den **I_WBSD** initialisiert (`INIT`) als auch den Digitalausgang aktualisiert (`REQ`).
4. **Wertausgabe**:
   - Der initialisierte **I_WBSD** gibt die aktuelle Geschwindigkeit (16-Bit) über `WHEELBASEDMACHINESPEED` aus.
   - Dieser Wert wird im Baustein **F_UINT_TO_UDINT** in einen 32-Bit-Wert konvertiert.
   - Das Bestätigungsereignis `CNF` des Konverters löst den **Q_NumericValue** aus (`REQ`), der den umgewandelten Wert übernimmt (`u32NewValue`) und auf dem UT anzeigt.

Die Übung veranschaulicht somit die kombinierte Nutzung von Hardware-Eingang, Flipflop-Logik und ISOBUS-Kommunikation zur Erfassung und Anzeige einer Maschinengröße.

Zusammenfassung

Die Übung **Uebung_070b** zeigt, wie ein digitaler Taster (I2) verwendet wird, um über einen T-Flipflop die Freigabe (QI) der radbasierten Geschwindigkeit zu steuern. Die gemessene Geschwindigkeit wird auf einem ISOBUS-Universal Terminal dargestellt. Der Aufbau ist konsistent mit dem Prinzip der Übung **Uebung_094a**. Das Verständnis dieser Schaltung ist grundlegend für Anwendungen, bei denen Maschinendaten zyklisch oder ereignisgesteuert ausgegeben werden sollen.

2.2.1.269 Uebung_070c: WBSD auf UT ausgeben, PHYS



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung zeigt, wie die radbasierte Maschinengeschwindigkeit (Wheel Based Machine Speed, kurz WBSD) von einem Feldbus gelesen, in einen physikalischen Wert umgerechnet und auf einem Universal Terminal (UT) als numerischer Wert angezeigt wird. Dabei wird der rohe Ganzzahlwert (UINT) mittels Skalierung in einen realen Zahlenwert (z. B. m/s) umgewandelt und an das UT übergeben.

• – ◦ • – ◦ • – ◦ •

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: I_WBSD

- **Typ**: `isobus::tecu::I_WBSD`
- **Verwendete interne FBs**: (keine)
  - **Parameter**:
    - `QI` = `TRUE` (aktiviert den Baustein)
  - **Ereignisausgang/-eingang**:
    - Ereignisausgang `IND` – meldet einen neuen gültigen Wert
  - **Datenausgang/-eingang**:
    - Datenausgang `WHEELBASEDMACHINESPEED` (UINT) – Rohwert der Radgeschwindigkeit
- **Funktionsweise**: Der Baustein liest über den ISOBUS-Feldbus den aktuellen Wert der radbasierten Maschinengeschwindigkeit (WBSD) ein. Bei einem neuen, gültigen Messwert wird das Ereignis `IND` ausgelöst.

Sub-Bausteine: FIELDBUS_UINT_TO_SIGNAL_SCALED

- **Typ**: `logiBUS::signalprocessing::fieldbus::FIELDBUS_UINT_TO_SIGNAL_SCALED`
- **Verwendete interne FBs**: (keine)
  - **Parameter**:
    - `SCALE` = `0.001`
    - `OFFSET` = `0`
  - **Ereigniseingang/-ausgang**:
    - Ereigniseingang `REQ` – startet die Umwandlung
    - Ereignisausgang `CNF` – bestätigt Abschluss
  - **Dateneingang/-ausgang**:
    - Dateneingang `IN` (UINT) – Rohwert
    - Datenausgang `OUT` (REAL) – umgerechneter skalierter Wert

- **Funktionsweise**: Der Baustein wandelt einen vorzeichenlosen 16-Bit-Integer (UINT) in eine reelle Zahl um. Die Umrechnungsformel lautet: `OUT = IN * SCALE + OFFSET`. Mit `SCALE = 0.001` und `OFFSET = 0` werden z. B. mm/s in m/s konvertiert.

Sub-Bausteine: Q_NumericValue

- **Typ**: `isobus::UT::Q::Q_NumericValue_PHYS`
- **Verwendete interne FBs**: (keine)
  - **Parameter**:
    - `stObj = NumberVariable_Wheel_based_machine_speed` (Verweis auf die Objektdefinition im UT-Pool)
  - **Ereigniseingang/-ausgang**:
    - Ereigniseingang `REQ` – aktualisiert den Wert auf dem UT
  - **Dateneingang/-ausgang**:
    - Dateneingang `rPhys` (REAL) – physikalischer Wert, der auf dem UT angezeigt wird
- **Funktionsweise**: Der Baustein stellt den übergebenen physikalischen Wert (REAL) über den ISOBUS-UT-Standard auf dem Universal Terminal dar. Die konkrete Darstellung (z. B. Einheit, Nachkommastellen) wird durch die im Pool referenzierte Objektdefinition `NumberVariable_Wheel_based_machine_speed` festgelegt.

• – ◦ • – ◦ • – ◦ •

Programmablauf und Verbindungen

Die drei Funktionsbausteine sind über Ereignis- und Datenverbindungen zu einer Kaskade verknüpft:

1. **Ereigniskette** `I_WBSD.IND` ⟶ `FIELDBUS_UINT_TO_SIGNAL_SCALED.REQ` ⟶ `FIELDBUS_UINT_TO_SIGNAL_SCALED.CNF` ⟶ `Q_NumericValue.REQ`
   - Der Feldbus-Baustein erzeugt bei einem neuen Radgeschwindigkeitswert das Ereignis `IND`, das die Umrechnung anstößt.
   - Nach erfolgreicher Umrechnung signalisiert `CNF` dem UT-Baustein, den aktualisierten Wert anzuzeigen.
2. **Datenfluss** `I_WBSD.WHEELBASEDMACHINESPEED` ⟶ `FIELDBUS_UINT_TO_SIGNAL_SCALED.IN` `FIELDBUS_UINT_TO_SIGNAL_SCALED.OUT` ⟶ `Q_NumericValue.rPhys`
   - Der Rohwert (UINT) wird direkt an den Umrechner weitergeleitet.
   - Das skalierte Ergebnis (REAL) wird dem UT-Baustein als physikalischer Wert übergeben.

Lernziele:

- Verständnis der ISOBUS-Datenschnittstellen für Geschwindigkeitssignale.
- Nutzung eines Skalierungsbausteins zur Umrechnung von Ganzzahl- auf physikalische Werte.
- Darstellung von Prozesswerten auf einem Universal Terminal.

Schwierigkeitsgrad: Fortgeschritten (Grundkenntnisse in ISOBUS und 4diac-IDE erforderlich)

Starten der Übung: Importieren Sie die SubApp in Ihr 4diac-Projekt und binden Sie sie in eine geeignete Anwendung (z. B. mit einem eventgetriebenen Zyklus) ein. Stellen Sie sicher, dass die referenzierte UT-Objektdefinition `NumberVariable_Wheel_based_machine_speed` im entsprechenden Pool vorhanden ist.

• – ◦ • – ◦ • – ◦ •

Zusammenfassung

Die Übung **Uebung_070c** demonstriert einen vollständigen Datenpfad von der Feldbuserfassung (WBSD) über die skalierte Umrechnung bis hin zur Anzeige auf einem Universal Terminal. Die Verwendung standardisierter ISOBUS-Bausteine ermöglicht eine einfache Integration in landtechnische Steuerungssysteme und zeigt, wie physikalische Werte aus rohen Busdaten gewonnen werden können.

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

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

2.2.1.270 Uebung_071: WBSD auf UT ausgeben, Q1 Schalten wenn Speed >0

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_071`. Hier wird die Traktor-Geschwindigkeit nicht nur angezeigt, sondern direkt zur Steuerung eines Aktors verwendet.

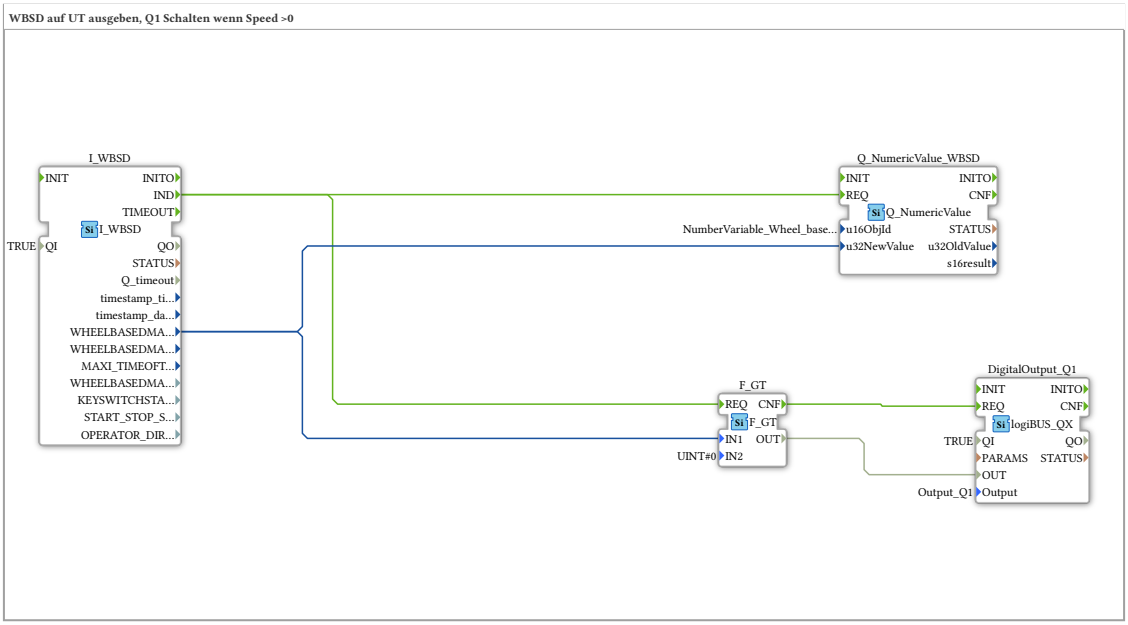
Ziel der Übung

Implementierung einer Schwellwert-Logik basierend auf TECU-Daten. Der Ausgang soll automatisch aktiviert werden, sobald sich die Maschine in Bewegung setzt.

Beschreibung und Komponenten

In `Uebung_071.SUB` wird die radbasierte Geschwindigkeit mit einem festen Wert verglichen.

Funktionsbausteine (FBs)



- **`I_WBSD`**: Liefert die aktuelle Geschwindigkeit.
- **`F_GT`**: Ein Vergleichs-Baustein (Greater Than). Er prüft, ob der Eingangswert größer als 0 ist.
- **`DigitalOutput_Q1`**: Der Hardware-Ausgang.

Funktionsweise

Die Logik reagiert auf jede Geschwindigkeits-Nachricht der TECU:

1. `I_WBSD.IND` triggert den Vergleich `F_GT`.
2. Ist die Geschwindigkeit > 0, liefert `F_GT.OUT` ein `TRUE`.
3. Das Bestätigungs-Event `CNF` fordert den Ausgang `Q1` zur Aktualisierung auf.

Ergebnis: Sobald der Traktor anfährt, schaltet der Ausgang `Q1` ein. Bleibt er stehen (Speed = 0), schaltet der Ausgang sofort wieder aus.

Anwendungsbeispiel

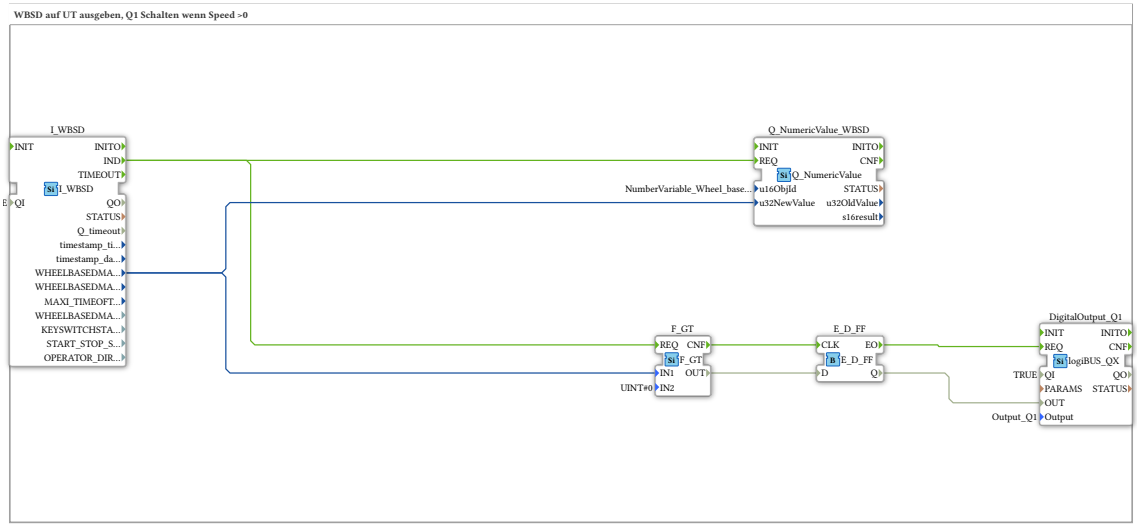
Automatischer Arbeitsstrahler: Eine Rückfahrkamera oder ein Zusatzscheinwerfer soll nur dann aktiv sein, wenn sich die Maschine tatsächlich bewegt. Dies spart Energie und verhindert die Blendung anderer Verkehrsteilnehmer im Stand.

2.2.1.271 Uebung_071a: WBSD auf UT ausgeben, Q1 Schalten wenn Speed >0

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_071a`.

Übersicht

Diese Variante der Übung 071 nutzt ein D-Flip-Flop (`E_D_FF`) zur zeitlichen Synchronisation des Schaltsignals. Das Ergebnis des Geschwindigkeitsvergleichs wird erst beim Eintreffen des Bestätigungs-Events am Takteingang des Flip-Flops fest übernommen und an den Ausgang `Q1` weitergegeben. Dies sorgt für ein stabileres Schaltverhalten in komplexeren Logik-Netzwerken, indem es sicherstellt, dass Daten und Ereignisse exakt im Gleichtakt verarbeitet werden.



2.2.1.272 Uebung_071b: WBSD auf UT ausgeben, Q1 Schalten wenn Speed >0

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_071b`. Hier steuern wir einen Ausgang nicht über die Geschwindigkeit, sondern über die zurückgelegte Wegstrecke.

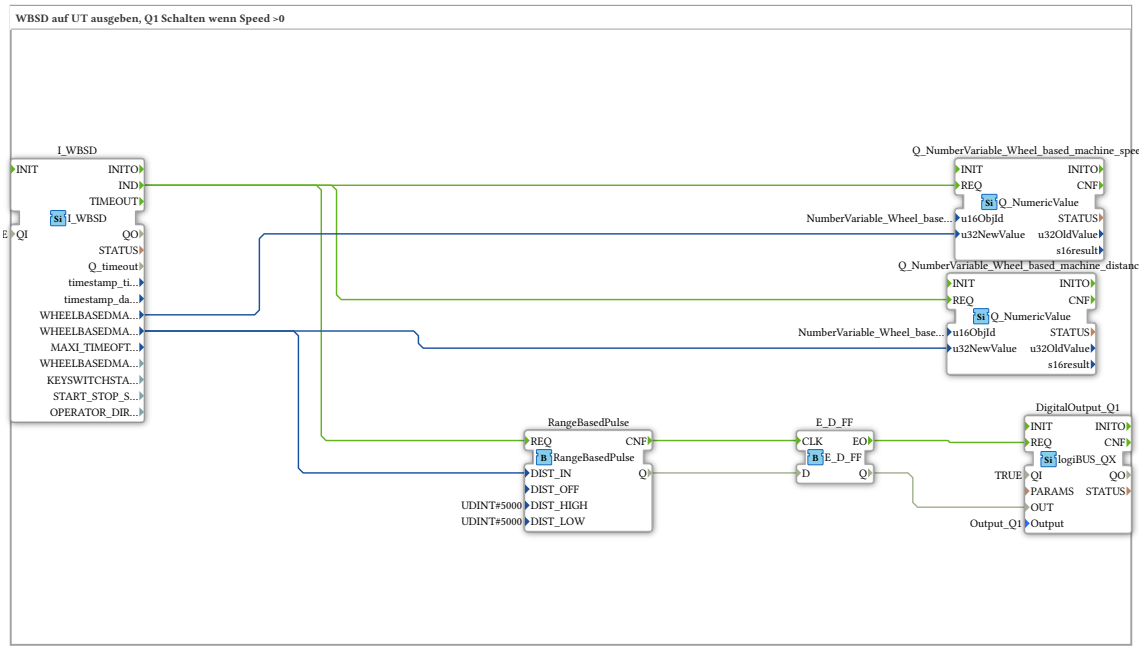
Ziel der Übung

Verwendung des Bausteins `RangeBasedPulse`. Es wird gezeigt, wie man ein periodisches Pulssignal erzeugt, das nicht zeitabhängig (alle X Sekunden), sondern wegabhängig (alle X Meter) ist.

Beschreibung und Komponenten

Die Subapplikation `Uebung_071b.SUB` liest die kumulierte Wegstrecke vom Traktor ein und erzeugt daraus Impulse.

Funktionsbausteine (FBs)



- **`I_WBSD`**: Liefert den Wert `WHEELBASEDMACHINEDISTANCE`.
- **`RangeBasedPulse`**: Dieser Baustein erzeugt einen Pegelwechsel am Ausgang `Q`, sobald eine definierte Distanz (hier 5000 mm = 5 Meter) überschritten wurde.
- **`E_D_FF`**: Synchronisiert den Puls für den Hardware-Ausgang.

Funktionsweise

1. Der Traktor fährt. Der Distanzwert am Baustein `I_WBSD` steigt kontinuierlich an.
2. Der `RangeBasedPulse` beobachtet diesen Wert.
3. Alle 5 Meter wechselt der Ausgang des Bausteins seinen Zustand.
4. Die Lampe an `Q1` blinkt also im Rhythmus des Weges: 5m An, 5m Aus, 5m An...

Anwendungsbeispiel

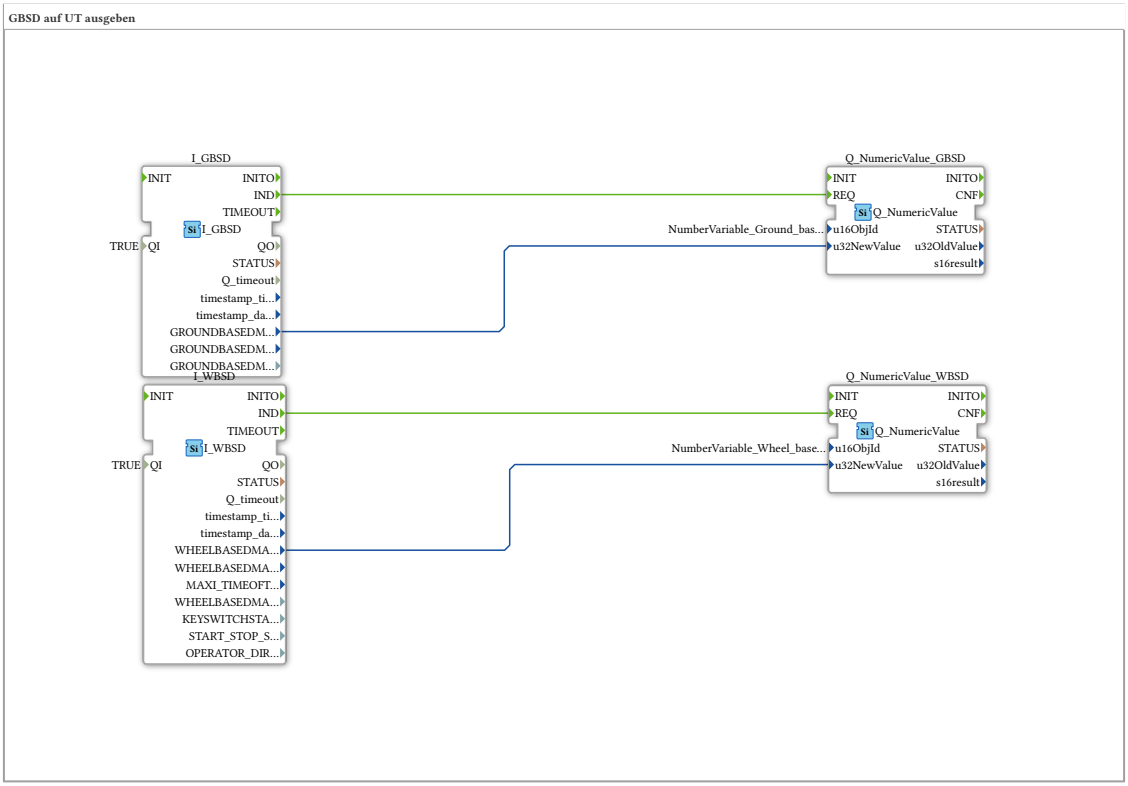
Wegabhängige Dosierung: Eine Sämaschine soll alle 10 Meter eine Bodenprobe markieren oder ein Farbsignal abgeben. Durch die Koppelung an den WBSD-Distanzwert erfolgt diese Markierung immer exakt im gleichen Abstand, egal wie schnell oder langsam der Traktor fährt.

2.2.1.273 Uebung_072: GBSD auf UT ausgeben

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_072`. In der Landtechnik gibt es verschiedene Quellen für die Geschwindigkeit; hier werden die zwei wichtigsten verglichen.

🎧 Podcast



- Eclipse 4diac 3.0: ST-Interpreter, FBE und 7200 Commits – Der Turbo für verteilte Automatisierung

---

Ziel der Übung

Gleichzeitige Verarbeitung von radbasierter (WBSD) und grundbasierter (GBSD) Geschwindigkeit.

Beschreibung und Komponenten

In `Uebung_072.SUB` werden zwei verschiedene TECU-Eingangsbausteine genutzt und deren Werte auf dem Terminal angezeigt.

Funktionsbausteine (FBs)

- **`I_WBSD`**: Radbasierte Geschwindigkeit (Wheel Based). Sie stammt meist vom Getriebe-Sensor.
- **`I_GBSD`**: Grundbasierte Geschwindigkeit (Ground Based). Sie wird meist über einen Radar-Sensor oder GPS-Empfänger ermittelt.

Hintergrund: Warum zwei Werte?

Auf losem Untergrund (z.B. nasser Acker) haben die Räder oft Schlupf. Die radbasierte Geschwindigkeit ist dann höher als die tatsächliche Vorwärtsbewegung. Die grundbasierte Geschwindigkeit (Radar) ist in diesem Fall genauer. Durch den Vergleich beider Werte im Programm kann die Steuerung den **Schlupf** berechnen und die Arbeitsprozesse entsprechend anpassen.

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

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

2.2.1.274 Uebung_072b: GBSD auf UT ausgeben, mit Impulsen



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_072b`. Hier wird eine komplexe wegabhängige Steuerung für mehrere Ausgänge realisiert.

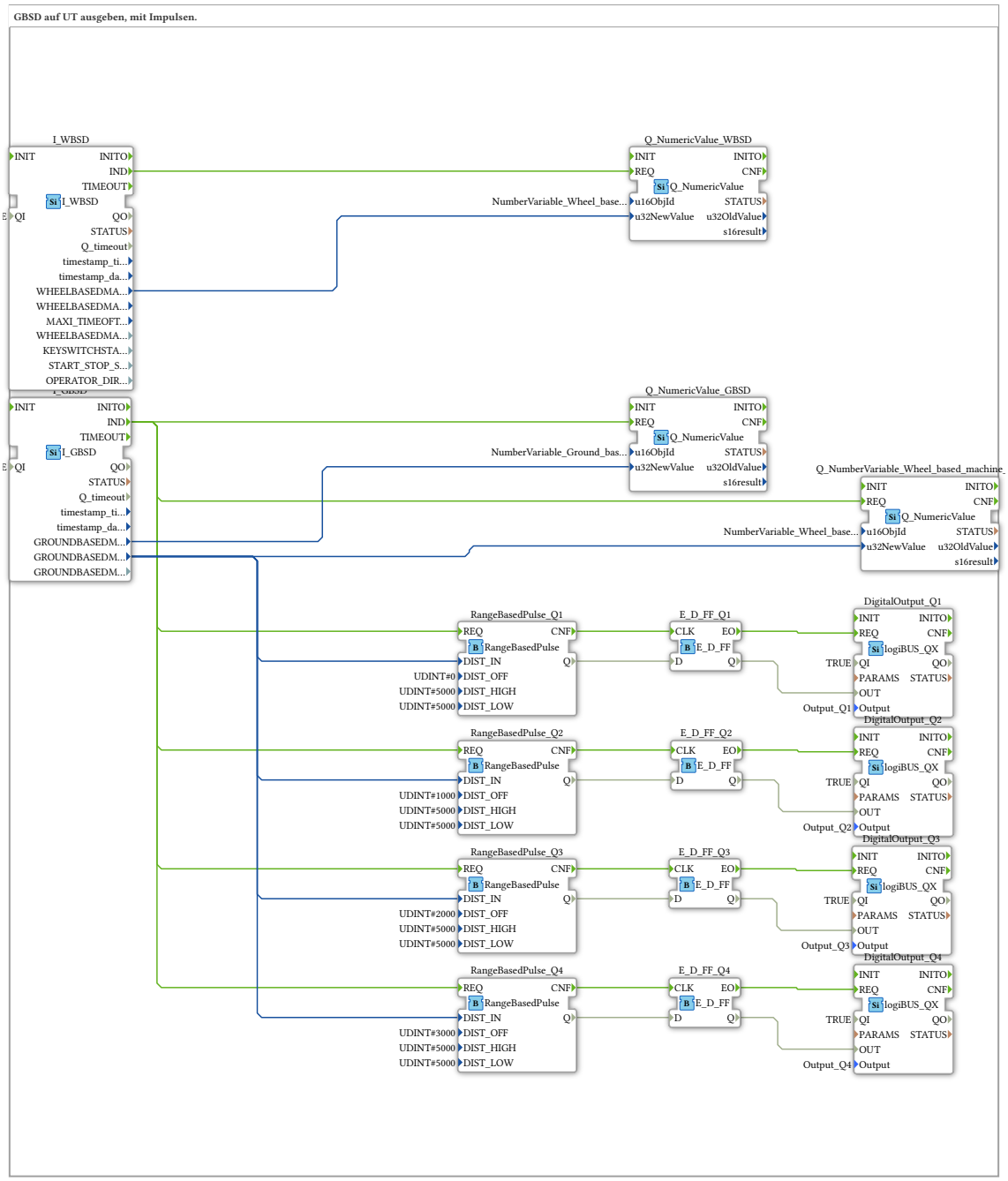
Ziel der Übung

Erzeugung von zeitversetzten Impulsen basierend auf dem GBSD-Distanzwert.

Beschreibung und Komponenten

In `Uebung_072b.SUB` steuern vier `RangeBasedPulse` Bausteine vier Ausgänge (`Q1` bis `Q4`) an.

Funktionsweise



Alle Bausteine reagieren auf den gleichen Distanzwert vom Radar (`I_GBSD`). Sie unterscheiden sich jedoch im Parameter **`DIST_OFF`** (Offset):

- `Q1`: Offset 0 mm.
- `Q2`: Offset 1000 mm.

- `Q3`: Offset 2000 mm.
- `Q4`: Offset 3000 mm.

Dies bewirkt ein „Wander-Muster“: Wenn die Maschine fährt, schalten die Ausgänge nacheinander ein und aus, jeweils um einen Meter versetzt zur gefahrenen Strecke.

Anwendungsbeispiel

Reihen-Steuerung bei Sämaschinen: Die Sähaggregate sind mechanisch versetzt am Rahmen montiert. Um exakt auf einer Linie quer zur Fahrtrichtung mit der Ablage zu beginnen, müssen die Aggregate je nach Fahrgeschwindigkeit und Position zeitversetzt angesteuert werden. Die Offset-Logik sorgt dafür, dass jedes Aggregat genau an der richtigen Stelle im Feld aktiv wird.

2.2.1.275 Uebung_072c: GBSD auf UT ausgeben



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_072c`. Hier wird eine mathematische Methode gezeigt, um aus der Geschwindigkeit die zurückgelegte Strecke selbst zu berechnen (Integration).

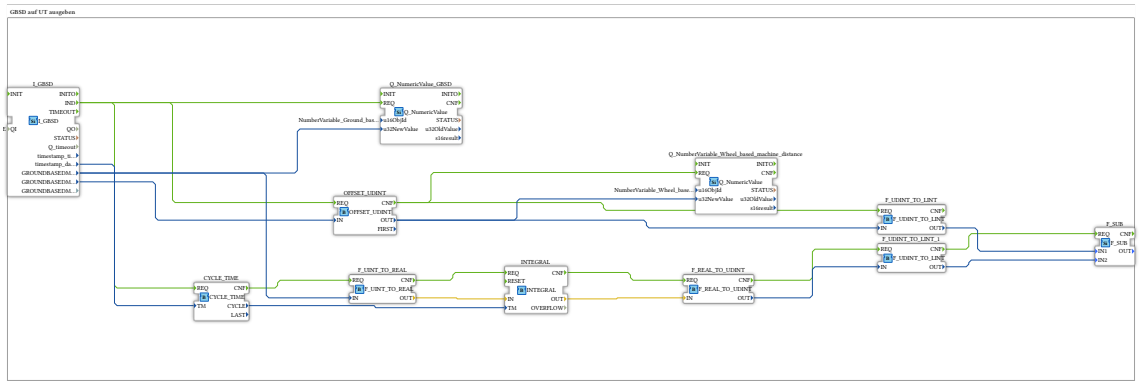
Ziel der Übung

Verwendung des Bausteins `INTEGRAL`. Es wird demonstriert, wie man einen Wegwert manuell berechnet, falls die TECU keinen kumulierten Distanzwert liefert oder dieser für eine Teilmessung (Tageskilometerzähler) genullt werden soll.

Beschreibung und Komponenten

Die Subapplikation `Uebung_072c.SUB` berechnet den Weg durch zeitliche Integration der radarbasierten Geschwindigkeit.

Funktionsbausteine (FBs)

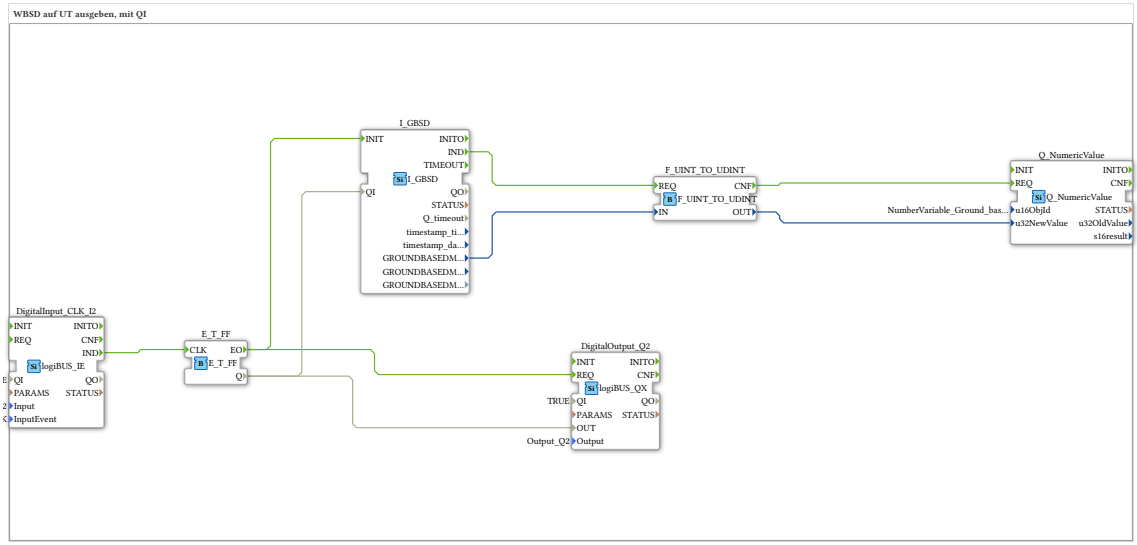


- **`I_GBSD`**: Liefert die aktuelle Geschwindigkeit.
- **`CYCLE_TIME`**: Misst die Zeit zwischen zwei Geschwindigkeits-Nachrichten (`TM`).
- **`INTEGRAL`**: Summiert das Produkt aus Geschwindigkeit und Zeit (`v * dt`) auf.
- **`OFFSET_UDINT`**: Erlaubt das Addieren eines Startwerts oder das Zurücksetzen der Zählung.

Funktionsweise

Das Programm führt permanent die physikalische Grundformel `Weg = Geschwindigkeit * Zeit` aus. Da die TECU-Daten nie ganz glatt sind, nutzt der `INTEGRAL` Baustein kleine Zeitintervalle (`CYCLE_TIME`), um eine hohe Genauigkeit der aufsummierten Strecke zu erreichen. Das Ergebnis wird am Terminal als `Wheel_based_machine_distance` angezeigt (auch wenn es hier aus Radar-Daten berechnet wurde).

2.2.1.276 Uebung_072d: WBSD auf UT ausgeben, mit QI



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung zeigt, wie die arbeitsbreitenbezogene Maschinengeschwindigkeit (WBSD – Working Width Based Ground Speed) auf einem Universal Terminal (UT) ausgegeben werden kann. Die Ausgabe erfolgt gesteuert über einen Qualitätsindikator (QI), der über einen Taster ein- und ausgeschaltet wird. Der QI bestimmt, ob die aktuelle Geschwindigkeit an das UT gesendet wird oder nicht. Der Taster (Digital Input) steuert über einen T-Flipflop sowohl den QI als auch einen digitalen Ausgang (Q2) zur Statusanzeige.

Verwendete Funktionsbausteine (FBs)

- **I_GBSD** (Typ: `isobus::tecu::I_GBSD`)
  - Liest die arbeitsbreitenbezogene Maschinengeschwindigkeit (Ground Based Machine Speed) aus dem CAN-Bus.
- **F_UINT_TO_UDINT** (Typ: `iec61131::conversion::F_UINT_TO_UDINT`)
  - Wandelt den 16-Bit-Wert (UINT) der Geschwindigkeit in einen 32-Bit-Wert (UDINT) um, wie er vom UT-Baustein erwartet wird.
- **Q_NumericValue** (Typ: `isobus::UT::Q::Q_NumericValue`)
  - Sendet einen numerischen Wert (hier die Geschwindigkeit) an das UT.
  - Parameter: `u16ObjId` = `NumberVariable_Ground_based_machine_speed` (definiert in der importierten Pool-Datei).
- **E_T_FF** (Typ: `iec61499::events::E_T_FF`)
  - T-Flipflop: Bei jedem positiven Takt (Ereignis am CLK) wechselt der Ausgang Q seinen Zustand.
- **DigitalInput_CLK_I2** (Typ: `logiBUS::io::DI::logiBUS_IE`)
  - Erfasst ein Tasterereignis am Digitaleingang I2 (Konfiguration: `Input_I2`, Ereignis `BUTTON_SINGLE_CLICK`).
  - Parameter: `QI` = `TRUE`.
- **DigitalOutput_Q2** (Typ: `logiBUS::io::DQ::logiBUS_QX`)
  - Steuert den Digitalausgang Q2 zur optischen Rückmeldung des QI-Zustands.
  - Parameter: `QI` = `TRUE`, `Output` = `Output_Q2`.

Programmablauf und Verbindungen

1. **Eingangssignal**
   - Ein Tastendruck am Digitaleingang I2 erzeugt ein Ereignis (`BUTTON_SINGLE_CLICK`).
   - Dieses Ereignis wird über `DigitalInput_CLK_I2.IND` an `E_T_FF.CLK` weitergeleitet.
   - Der T-Flipflop `E_T_FF` ändert bei jedem Tastendruck seinen Ausgang `Q`.
2. **Steuerung des QI und des Digitalausgangs**
   - Der Zustand `Q` des Flipflops wird zwei Datenverbindungen zugeführt:
     - An `I_GBSD.QI` (Qualitätsindikator des Geschwindigkeitssensors).

  - An `DigitalOutput_Q2.OUT` (schaltet den Ausgang Q2 ein/aus).

3. **Geschwindigkeitserfassung und -umwandlung**
   - Solange `QI` = `TRUE` ist, sendet `I_GBSD` bei jeder Aktualisierung ein Ereignis `IND` aus.
   - Dieses Ereignis triggert `F_UINT_TO_UDINT.REQ`, welches die 16-Bit-Geschwindigkeit (`I_GBSD.GROUNDBASEDMACHINESPEED`) in einen 32-Bit-Wert wandelt.
   - Der umgewandelte Wert (`F_UINT_TO_UDINT.OUT`) wird an `Q_NumericValue.u32NewValue` übergeben.
4. **Ausgabe auf das UT**
   - Nach erfolgreicher Umwandlung erzeugt `F_UINT_TO_UDINT` das Ereignis `CNF`, das `Q_NumericValue.REQ` auslöst.
   - Der Baustein `Q_NumericValue` sendet den aktuellen Geschwindigkeitswert an das UT unter der Objekt-ID `NumberVariable_Ground_based_machine_speed`.
5. **Hinweis**
   - Die Übung **Uebung_094a** nutzt dasselbe Prinzip.

Zusammenfassung

Die Übung demonstriert die Verknüpfung von ISOBUS-Elementen (Geschwindigkeitssensor, UT-Ausgabe) mit Logikbausteinen (T-Flipflop, Digitalein-/ausgänge) in 4diac-IDE. Durch den Taster wird der Qualitätsindikator des Geschwindigkeitssignals gesteuert, sodass die Anzeige am UT nur bei aktiviertem QI aktualisiert wird. Ein digitaler Ausgang gibt den Zustand optisch wieder. Dies ermöglicht eine bedarfsgesteuerte Übertragung von Prozessdaten an das Terminal.

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

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

2.2.1.277 Uebung_073: VDS auf UT ausgeben



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_073`. Hier wird die dritte Geschwindigkeitsquelle des ISOBUS erschlossen: Das Navigationssystem.

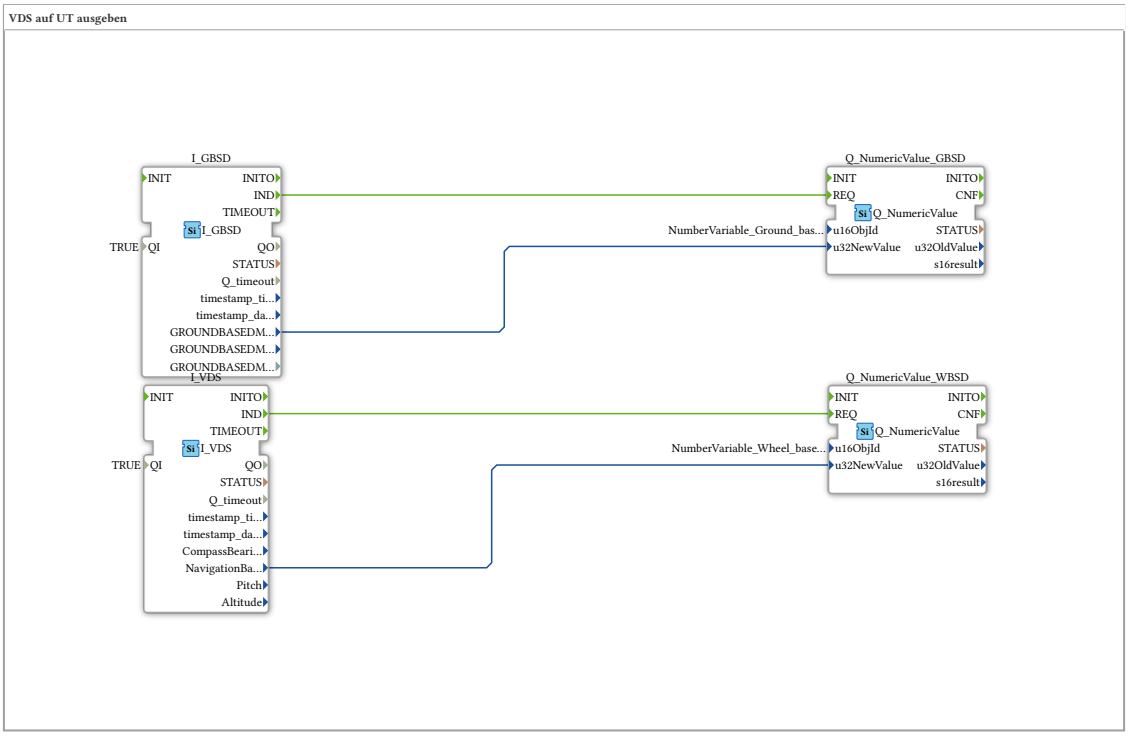
Ziel der Übung

Verwendung des Bausteins `I_VDS` (Vehicle Direction and Speed).

Beschreibung und Komponenten

In `Uebung_073.SUB` werden die Radar-Geschwindigkeit (GBSD) und die GPS-Geschwindigkeit (VDS) parallel verarbeitet.

Funktionsbausteine (FBs)



- **`I_VDS`**: Dieser Baustein empfängt Daten vom GPS-Empfänger des Traktors (`NavigationBasedVehicleSpeed`).

Funktionsweise

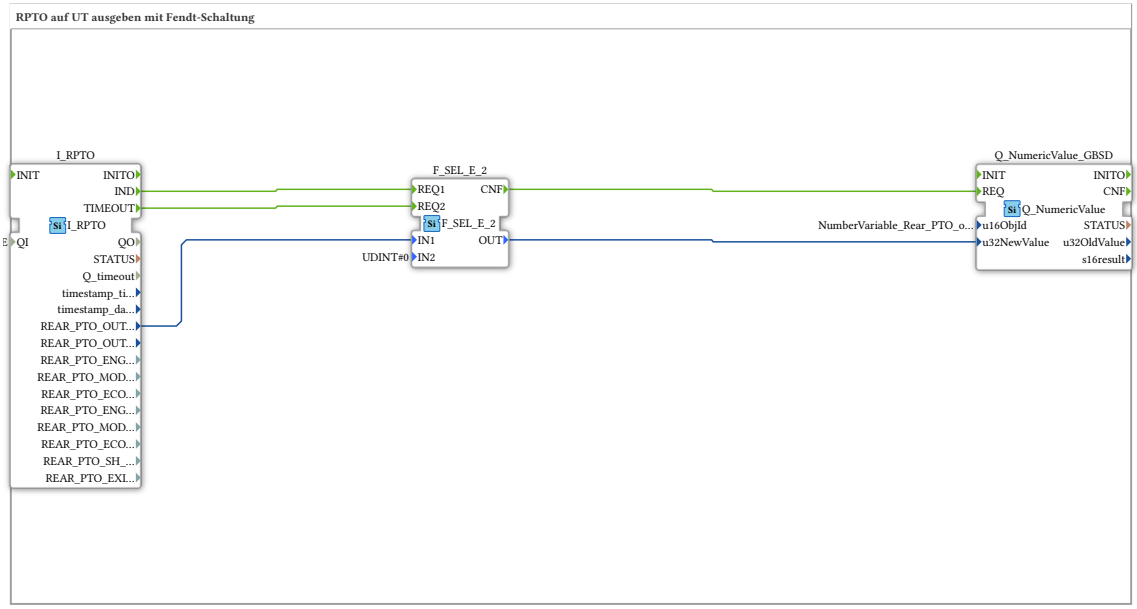
GPS-Daten sind besonders genau bei konstanter Fahrt auf freiem Feld, können aber bei schneller Beschleunigung oder unter Bäumen/an Gebäuden ungenau werden. In modernen Systemen nutzt man VDS oft als Referenz, um Radar-Sensoren zu kalibrieren oder bei deren Ausfall eine Ausweich-Geschwindigkeit zu haben.

2.2.1.278 Uebung_074: RPTO auf UT ausgeben

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_074`. Hier wird die Drehzahl der Heck-Zapfwelle (Power Take-Off) eingelesen.

🎧 Podcast



- Verpolungsschutz in der Elektronik: Warum die ideale Diode (LM74700) MOSFETs und Schottky-Dioden in Effizienz und Kosten schlägt

---

Ziel der Übung

Verwendung des Bausteins `I_RPTO` (Rear PTO). Es wird gezeigt, wie man mit einer Besonderheit mancher Traktoren umgeht: Wenn die Zapfwelle steht, senden einige TECUs keine „Null“, sondern hören einfach auf, Nachrichten zu schicken.

Beschreibung und Komponenten

In `Uebung_074.SUB` wird ein Sicherheits-Selektor verwendet, um eine saubere Null-Anzeige zu garantieren.

Funktionsbausteine (FBs)

- **`I_RPTO`**: Liefert die Drehzahl am Ausgang `REAR_PTO_OUTP_SHAFT_SPEED`.
- **`F_SEL_E_2`**: Wählt zwischen dem Messwert und einer festen Null aus.

Funktionsweise („Fendt-Schaltung“)

1. **Normalbetrieb**: Die TECU sendet Drehzahlen. `I_RPTO.IND` triggert den ersten Eingang des Selektors
   ➡️ Der Messwert wird zum Terminal durchgereicht.
2. **Stillstand**: Bleiben die Nachrichten der TECU für längere Zeit aus, feuert der Baustein `I_RPTO.TIMEOUT`.
3. **Sicherheit**: Dieses Timeout-Event triggert den zweiten Eingang des Selektors. Da hier die Konstante `0` anliegt, springt die Anzeige am Terminal sofort auf „0 U/min“ zurück. Dies verhindert, dass der letzte gemessene Wert (z.B. „540“) dauerhaft am Display stehen bleibt, obwohl die Welle bereits steht.

Anwendungsbeispiel

Gerätesteuerung mit Zapfwellen-Freigabe: Ein Gülle-Rührwerk darf nur arbeiten, wenn die Zapfwelle mindestens 300 U/min erreicht hat. Die Logik nutzt den `RPTO`-Wert zur Freigabe. Durch den Timeout-Schutz wird sichergestellt, dass die Freigabe sofort entzogen wird, sobald die Zapfwelle (und damit die TECU-Nachricht) stoppt.

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

- 🌐 Diode & Halbleitergrundlagen auf ms-muc-docs.de

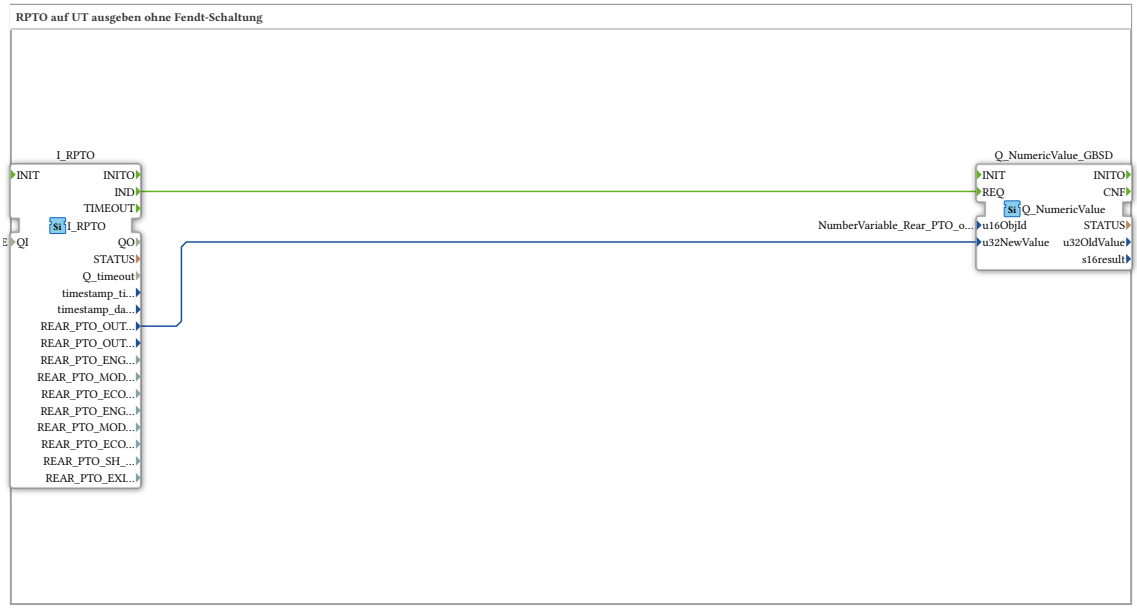
2.2.1.279 Uebung_074b: RPTO auf UT ausgeben ohne Fendt-Schaltung

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_074b`. Direkte (nicht-Adapter-) Variante von `Uebung_074b_AUI` - siehe dort für die konzeptionell identische Funktionsweise.

Ziel der Übung

Anzeige der Zapfwellendrehzahl (`Rear PTO output shaft speed`) auf dem VT-Terminal, direkt vom TECU-Wert abgeleitet, ohne Umschaltlogik.

Beschreibung und Komponenten

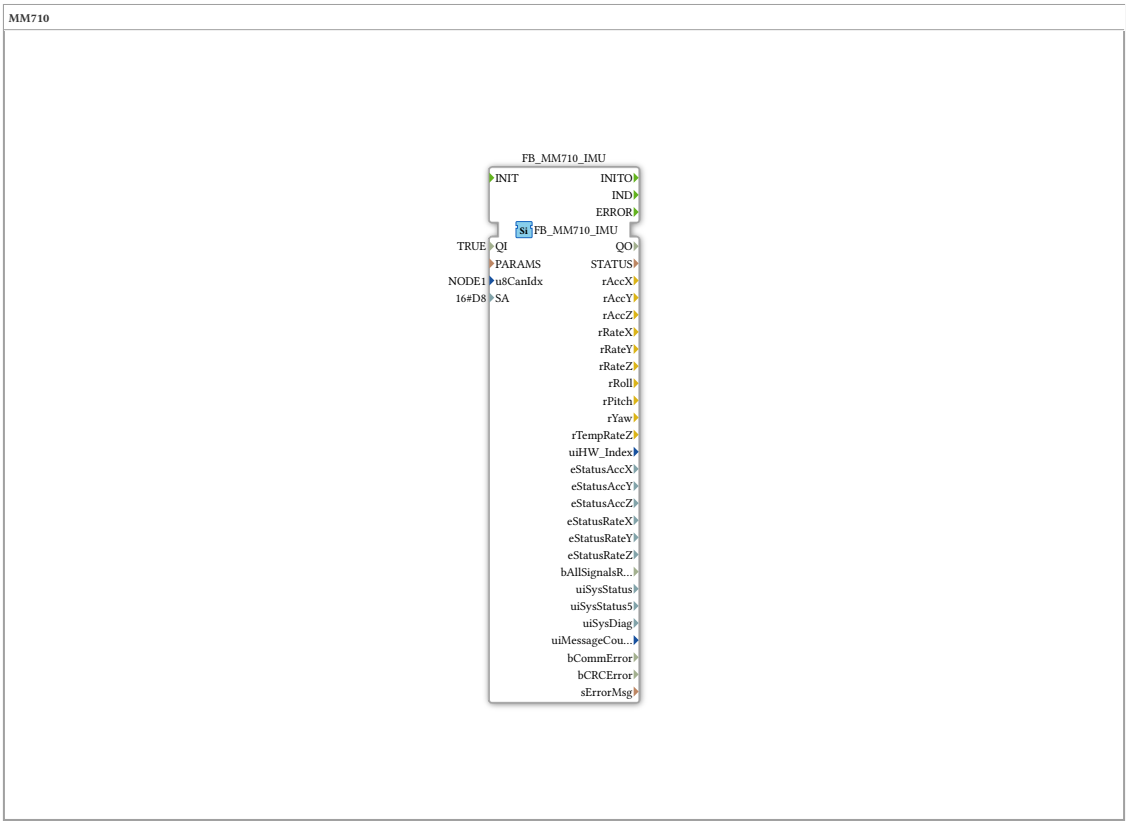


- **`I_RPTO`**: `isobus::tecu::I_RPTO`, liest die Zapfwellendrehzahl vom Tractor-ECU (TECU) über ISOBUS.
- **`Q_NumericValue_GBSD`**: zeigt den Wert direkt auf dem VT-Terminal an (`u16ObjId = NumberVariable_Rear_PTO_output_shaft_speed`).

Funktionsweise

1. `I_RPTO` liest laufend die Zapfwellendrehzahl vom TECU.
2. Bei jeder Meldung (`IND`) wird der Wert (`REAR_PTO_OUTP_SHAFT_SPEED`) direkt in `Q_NumericValue_GBSD` angezeigt - ohne Adapter- oder Auswahllogik dazwischen.

2.2.1.280 Uebung_075: MM710



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung demonstriert die Einbindung des Funktionsbausteins **FB_MM710_IMU** zur Kommunikation mit einem IMU-Sensor vom Typ MM710 über den logiBUS. Der Sensor wird über den CAN-Bus angesteuert und liefert Bewegungsdaten. Die Übung dient als Grundlage für das Verständnis der Parametrierung und Verwendung von CAN-basierten Sensorbausteinen in 4diac.

Verwendete Funktionsbausteine (FBs)

- **FB_MM710_IMU** (Typ: `logiBUS::bosch::imu::FB_MM710_IMU`)
  - **Parameter**:
    - `QI` = `TRUE` (Aktivierung des Bausteins)
    - `u8CanIdx` = `NODE1` (CAN-Knoten, definiert durch den Import `isobus::pgn::ISO_CAN_NODE::NODE1`)
    - `SA` = `16#D8` (Quelladresse des Sensors im CAN-Netzwerk)
  - **Funktionsweise**: Der FB stellt die Schnittstelle zum IMU-Sensor her. Über den CAN-Bus werden Konfigurations- und Messdaten ausgetauscht. Die Parametrierung erfolgt über die angegebenen Werte. Der Baustein ist für den Einsatz im logiBUS-Ökosystem vorgesehen.

Programmablauf und Verbindungen

Die SubApp enthält **keine weiteren Funktionsbausteine oder Sub-Bausteine**. Der gesamte Funktionsumfang wird durch den einzelnen FB `FB_MM710_IMU` realisiert. Die SubApp besitzt keine eigenen Ein- oder Ausgänge – sie dient als gekapselte Einheit für die Integration des IMU-Sensors.

Ablauf:

1. Nach Aktivierung der SPS wird der FB mit `QI = TRUE` initialisiert.
2. Der Baustein versucht, über den CAN-Bus mit dem Sensor unter der Adresse `0xD8` zu kommunizieren.
3. Nach erfolgreicher Verbindung können Sensordaten (z. B. Beschleunigung, Drehrate) ausgelesen werden (abhängig von der Implementierung des FB).

Lernziele:

- Verstehen der Parametrierung eines CAN-basierten IMU-Sensors.
- Kennenlernen des logiBUS-Konzepts und der Verbindung zu 4diac.
- Einblick in die Verwendung von Funktionsbausteinen aus der `logiBUS::bosch::imu`-Bibliothek.

Voraussetzungen:

- Grundkenntnisse der 4diac-IDE und des Editors für SubApplikationen.
- Verständnis der CAN-Bus-Kommunikation (ISO 11783, PGN).

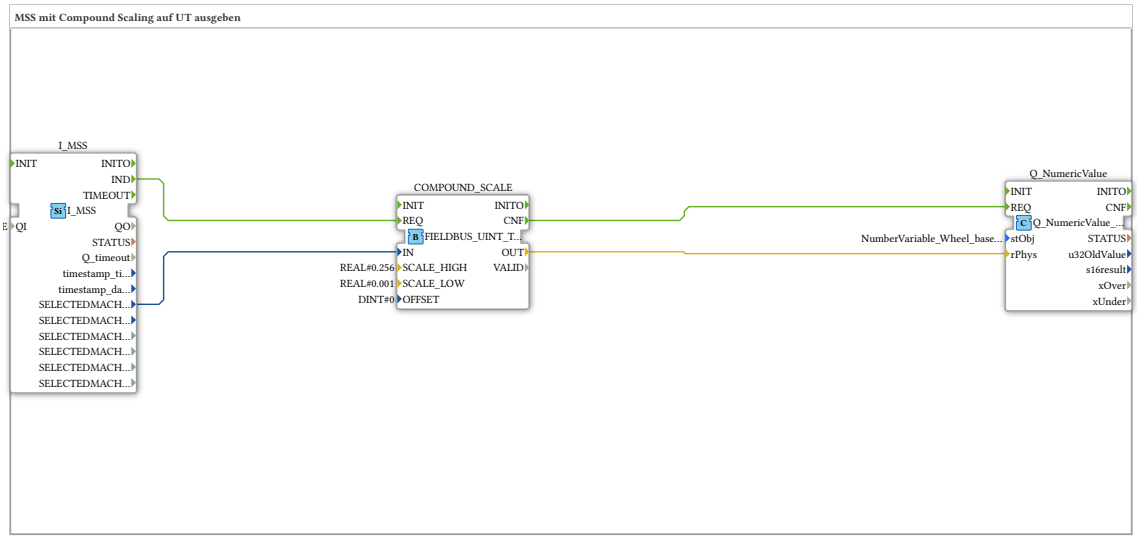
Zusammenfassung

Die Übung **Uebung_075** zeigt die grundlegende Integration des IMU-Sensors MM710 über den logiBUS. Der Funktionsbaustein `FB_MM710_IMU` wird mit den notwendigen Parametern konfiguriert und in einer SubApp gekapselt. Damit wird eine wiederverwendbare Komponente geschaffen, die in größere Automatisierungsprojekte eingebunden werden kann.

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

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

2.2.1.281 Uebung_076: MSS mit Compound Scaling auf UT ausgeben



• — ○ • — ○ • — ○ •

Einleitung

Diese Übung implementiert eine Funktion zur Verarbeitung einer Maschinengeschwindigkeit (Machine Selected Speed – MSS). Der Wert wird über einen Compound-Scale-Funktionsbaustein skaliert und anschließend auf ein Universal Terminal (UT) als numerischer Wert ausgegeben. Die Skalierung erfolgt mit einem oberen und unteren Faktor (0,256 und 0,001). Ein TODO-Hinweis weist darauf hin, dass der verwendete Objekt-Pool-Eintrag (NumberVariable_Wheel_based_machine_speed) vorerst als Platzhalter dient und später durch den korrekten Eintrag (NumberVariable_Machine_selected_speed) ersetzt werden sollte.

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: `I_MSS`

- **Typ**: `isobus::tecu::I_MSS`
- **Parameter**:
  - `QI` = `TRUE`
- **Funktionsweise**:
  Dieser FB stellt den Eingang für die Maschinengeschwindigkeit dar. Er liefert bei einer Eingabe an seinem Ereigniseingang `QI` den aktuellen Maschinengeschwindigkeitswert (Typ: vermutlich UINT) am Datenausgang `SELECTEDMACHINESPEED`. Der Ereignisausgang `IND` signalisiert, dass ein neuer Wert vorliegt.

Sub-Bausteine: `COMPOUND_SCALE`

- **Typ**: `logiBUS::signalprocessing::fieldbus::FIELDBUS_UINT_TO_SIGNAL_COMPOUND_SCALE`
- **Parameter**:
  - `SCALE_HIGH` = `REAL#0.256`
  - `SCALE_LOW` = `REAL#0.001`
  - `OFFSET` = `DINT#0`
- **Funktionsweise**:
  Dieser FB führt eine zusammengesetzte Skalierung (Compound Scaling) eines vorzeichenlosen Integer-Wertes (UINT) durch. Der eingehende Wert `IN` wird mit zwei verschiedenen Skalierungsfaktoren multipliziert, um eine höhere Genauigkeit im unteren und oberen Wertebereich zu erreichen. Der Ausgang `OUT` liefert den skalierten Wert (Typ: REAL). Der Ereigniseingang `REQ` startet die Berechnung; bei Abschluss wird am Ereignisausgang `CNF` ein Signal ausgegeben.

Sub-Bausteine: `Q_NumericValue`

- **Typ**: `isobus::UT::Q::Q_NumericValue_PHYS`
- **Parameter**:
  - `stObj` = `NumberVariable_Wheel_based_machine_speed`
- **Funktionsweise**:
  Dieser FB dient der Ausgabe eines numerischen Werts auf das Universal Terminal. Der übergebene physikalische Wert (Eingang `rPhys`, Typ: REAL) wird entsprechend der Eigenschaften des referenzierten Objekt-Pool-Eintrags (`stObj`) an das UT gesendet. Der Ereigniseingang `REQ` löst die Ausgabe aus.

Programmablauf und Verbindungen

Der Ablauf erfolgt rein intern innerhalb des Subapplikationstyps (keine äußeren Schnittstellen). Die Verbindungen zwischen den Funktionsbausteinen sind wie folgt:

Ereignisverbindungen:

1. `I_MSS.IND` ⟶ `COMPOUND_SCALE.REQ`
   Sobald ein neuer MSS-Wert anliegt, wird die Skalierung angestoßen.
2. `COMPOUND_SCALE.CNF` ⟶ `Q_NumericValue.REQ`
   Nach abgeschlossener Skalierung wird der ausgegebene Wert zum UT gesendet.

Datenverbindungen:

1. `I_MSS.SELECTEDMACHINESPEED` ⟶ `COMPOUND_SCALE.IN`
   Der rohe Maschinengeschwindigkeitswert (UINT) wird an den Skalierungsbaustein weitergeleitet.
2. `COMPOUND_SCALE.OUT` ⟶ `Q_NumericValue.rPhys`
   Der skalierte Wert (REAL) wird als physikalischer Wert für die UT-Ausgabe übergeben.

Ein Kommentar im Netzwerk enthält den Hinweis, dass der verwendete Objekt-Pool-Eintrag `NumberVariable_Wheel_based_machine_speed` nur als Platzhalter dient. In einer finalen Implementierung sollte dieser durch `NumberVariable_Machine_selected_speed` ersetzt werden.

Zusammenfassung

Die Übung `Uebung_076` demonstriert die Verarbeitung einer Maschinengeschwindigkeit mittels Compound Scaling und die anschließende Ausgabe auf ein Universal Terminal. Der Datenfluss ist linear: Der Eingangswert wird skaliert und dann als physikalische Größe an das UT übergeben. Die Skalierungsparameter sind fest vorgegeben (0,256 für den oberen, 0,001 für den unteren Bereich). Die Übung zeigt die Verwendung von ISOBUS-spezifischen Funktionsbausteinen (I_MSS, Q_NumericValue) sowie eines generischen Feldbus-Skalierungsbausteins. Der TODO-Kommentar weist auf eine notwendige Anpassung des Objekt-Pool-Eintrags hin.

2.2.1.282 Uebung_079: alle TECU FBs



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_079`. Dies ist eine Sammel-Übung, die alle verfügbaren Bausteine zur Erfassung von Traktor-Informationen vorstellt.

Ziel der Übung

Erlernen der gesamten Palette an TECU-Schnittstellenbausteinen. Ein ISOBUS-Traktor meldet eine Vielzahl von physikalischen Werten über den CAN-Bus, die in 4diac direkt als Bausteine genutzt werden können.

Übersicht der Bausteine (FBs)

In `Uebung_079.SUB` sind alle relevanten TECU-Eingangsbausteine platziert:



1. **`I_GBSD`**: Ground Based Speed & Distance (Radar/GPS-Weg).
2. **`I_WBSD`**: Wheel Based Speed & Distance (Getriebe-Weg).

3. **`I_VDS`**: Vehicle Direction and Speed (Navigationsdaten).
4. **`I_RPTO` & `I_FPTO`**: Heck- und Front-Zapfwellendrehzahl (Rear/Front Power Take-Off).
5. **`I_RHS` & `I_FHS`**: Heck- und Front-Hubwerksposition (Rear/Front Hitch Status).
6. **`I_MSS`**: Machine Specific Status.
7. **`COGSOGRapidUpdate`**: Hochfrequente Kurs- und Geschwindigkeitsdaten über Grund.

Anwendung in der Praxis

Jeder dieser Bausteine lauscht auf die standardisierten ISOBUS-Nachrichten der jeweiligen Traktor-ECU. Das logiBUS-System sorgt dafür, dass diese komplexen Protokoll-Daten in einfache IEC 61499 Ereignisse und Datenwerte gewandelt werden. Der Entwickler muss sich nicht um CAN-IDs oder Bit-Masken kümmern, sondern kann direkt mit den physikalischen Größen wie „Drehzahl“ oder „Position“ arbeiten.

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

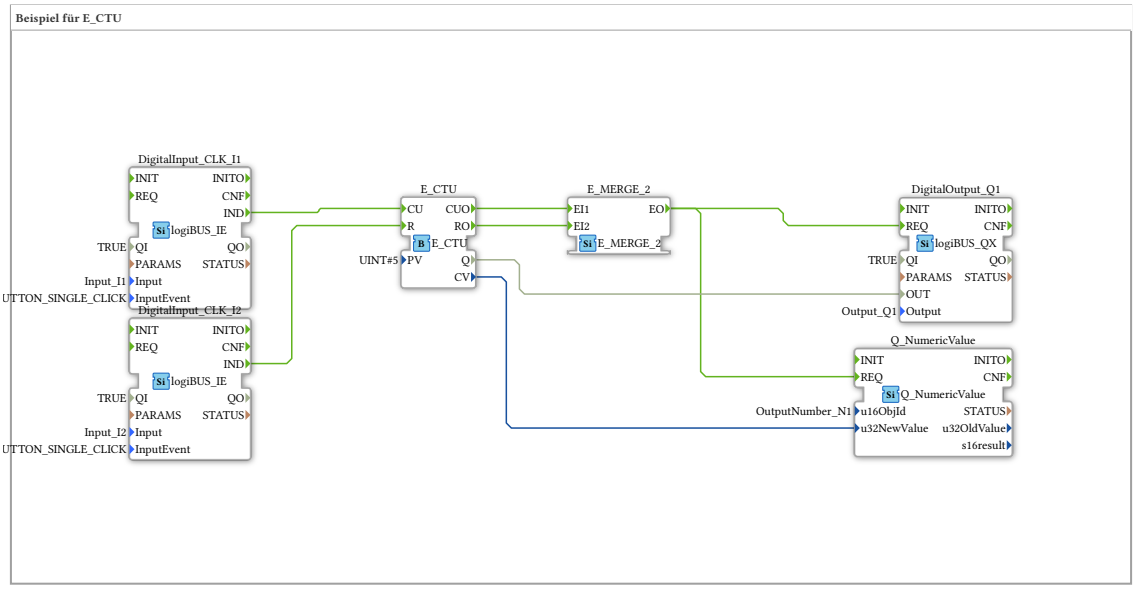
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de
- 🌐 IEC 61499 Events – Der Puls der Automatisierung auf ms-muc-docs.de

2.2.1.283 Uebung_080: Beispiel für E_CTU

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_080`. Hier wird das grundlegende Prinzip des Zählens von Ereignissen vorgestellt.

🎧 Podcast



- 800 PS Hightech-Riese: Was die Betriebsanleitung des ROPA Tiger 6S über moderne Landwirtschaft und extreme Sicherheit verrät

---

Ziel der Übung

Verwendung des Bausteins `E_CTU` (Event Count Up). Es wird gezeigt, wie man eine bestimmte Anzahl von Ereignissen (z.B. Tastendrücke) erfasst und beim Erreichen eines Grenzwerts eine Aktion auslöst.

Beschreibung und Komponenten

Die Subapplikation `Uebung_080.SUB` nutzt einen Zählerbaustein mit Set- und Reset-Logik.

Funktionsbausteine (FBs)

- **`DigitalInput_I1` (Count)**: Jeder Klick erhöht den Zähler.
- **`DigitalInput_I2` (Reset)**: Setzt den Zählerstand auf Null zurück.
- **`E_CTU`**: Der Zähler-Baustein. Der Parameter `PV` (Preset Value) ist auf 5 eingestellt.
- **`DigitalOutput_Q1`**: Zeigt den Status des Zählers an.

Funktionsweise

1. Der Nutzer klickt auf **I1**. Der Zählerstand (`CV`) erhöht sich bei jedem Event.
2. Der Ausgang `Q` des Zählers wechselt auf `TRUE`, sobald der Zählerstand den Wert 5 erreicht oder überschreitet (`CV >= PV`).
3. Die Lampe an **Q1** leuchtet auf.
4. Durch Klick auf **I2** wird der Zähler gelöscht, `Q` wird wieder `FALSE` und die Lampe geht aus.

Anwendungsbeispiel

Stückzähler: An einer Verpackungsmaschine werden die Kartons gezählt. Sobald 5 Kartons auf der Palette sind, wird ein Signal (`Q1`) gegeben, um die Palette automatisch auszufahren. Der Fahrer drückt nach dem Holen einer neuen Palette „Reset“ (`I2`), um den nächsten Vorgang zu starten.

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

- 🌐 E_CTU Event Counter Baustein auf ms-muc-docs.de

2.2.1.284 Uebung_080b: Beispiel für E_CTU mit E_SPLIT für Event-Verdoppelung und Terminal-Ausgabe



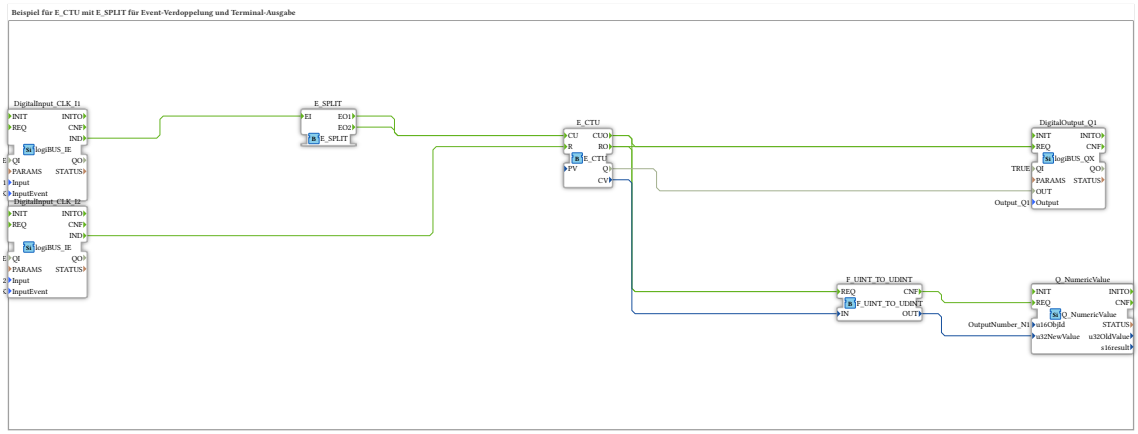
Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_080b`. Hier wird gezeigt, wie man die Anzahl der eintreffenden Ereignisse künstlich verdoppelt.

Ziel der Übung

Manipulation von Ereignisströmen unter Verwendung von `E_SPLIT`.

Funktionsweise

In `Uebung_080b.SUB` wird ein Ereignis-Splitter vor den Zähler geschaltet.



Jeder einzelne Klick auf Taster **I1** erreicht den Eingang `E_SPLIT.EI`. Der Splitter feuert daraufhin **zwei** Ereignisse (`EO1` und `EO2`) nacheinander ab. Da beide Ausgänge wieder auf den `CU`-Eingang des Zählers gemerged (zusammengeführt) werden, erhält der Zähler pro Tastendruck zwei Impulse.

Ergebnis: Die Lampe `Q1` (Schwelle 5) leuchtet bereits nach dem 3. Tastendruck auf (Zählerstand ist dann bereits auf 6 gesprungen).

Anwendungsbeispiel

Anpassung von Sensor-Impulsen: Ein Getriebesensor liefert einen Impuls pro Radumdrehung, die Logik benötigt aber zwei Impulse pro Umdrehung zur genaueren Berechnung. Der Splitter verdoppelt die eintreffende Frequenz rein softwareseitig.

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

- 🌐 E_CTU Event Counter Baustein auf ms-muc-docs.de

2.2.1.285 Uebung_080c: Beispiel für E_CTU mit Terminal-Ausgabe



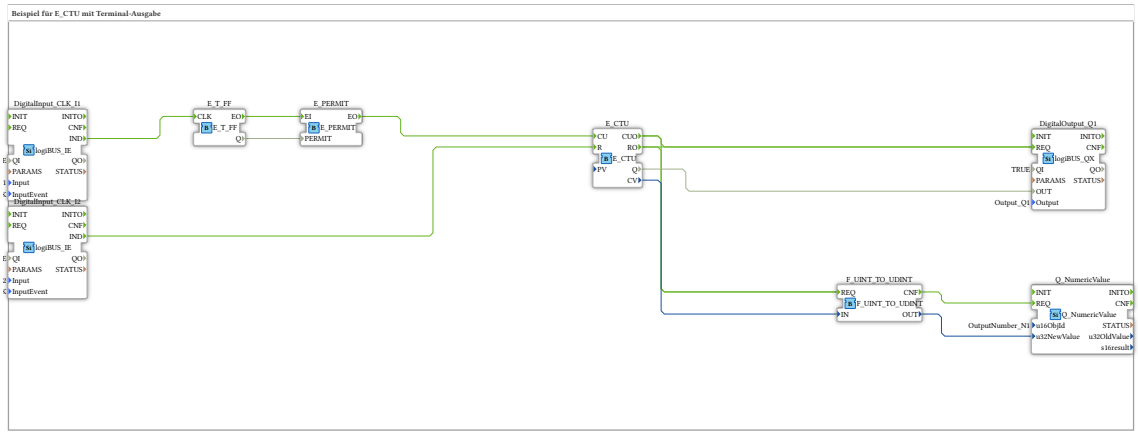
Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_080c`. Hier wird das Gegenteil der vorherigen Übung gezeigt: Die Reduzierung der Ereignisanzahl um die Hälfte.

Ziel der Übung

Manipulation von Ereignisströmen unter Verwendung von `E_T_FF` und `E_PERMIT`.

Funktionsweise

In `Uebung_080c.SUB` wird ein Toggle-Flip-Flop als Gate-Wächter genutzt.



1. Jeder Klick auf **I1** toggelt das Flip-Flop. Der Zustand wechselt also: TRUE, FALSE, TRUE, FALSE…
2. Das `E_PERMIT` lässt Ereignisse nur durch, wenn der `PERMIT` Eingang auf TRUE steht.
3. Daher wird nur bei jedem zweiten Klick (wenn das Flip-Flop gerade auf TRUE steht) ein Ereignis an den Zähler weitergereicht.

Ergebnis: Um die Lampe `Q1` (Schwelle 5) zum Leuchten zu bringen, muss der Nutzer nun 10 mal auf den Taster klicken.

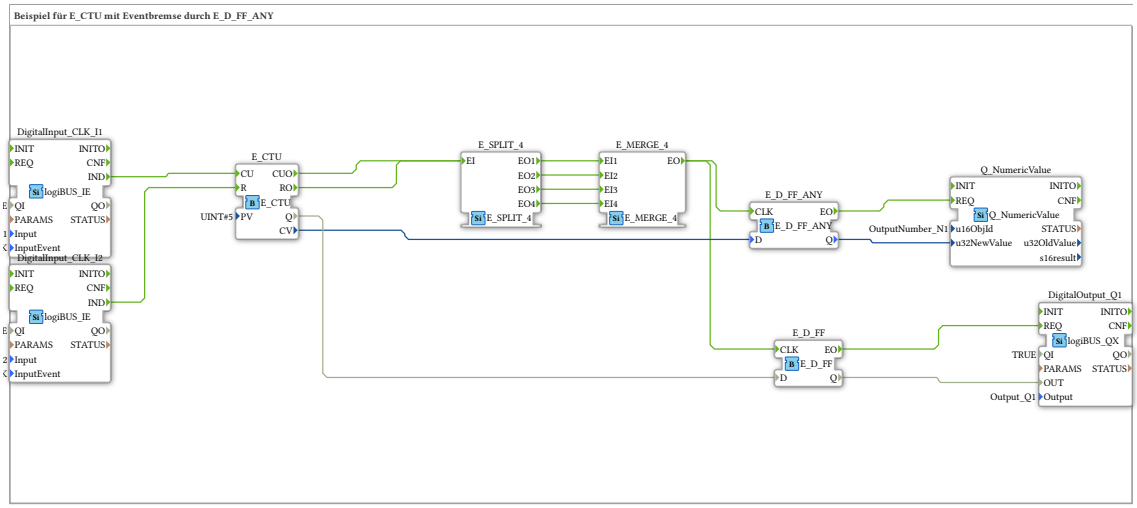
Anwendungsbeispiel

Unterdrückung von Prell-Effekten oder grobe Skalierung von schnellen Sensorsignalen, um die Rechenlast der nachfolgenden Logik zu verringern.

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

- 🌐 E_CTU Event Counter Baustein auf ms-muc-docs.de

2.2.1.286 Uebung_080d: Beispiel für E_CTU mit Eventbremse durch E_D_FF_ANY



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert den Einsatz eines Aufwärtszählers (`E_CTU`) in Kombination mit einer **Event-Bremse**, realisiert durch die beiden Flip-Flops `E_D_FF_ANY` und `E_D_FF`. Der Zählerstand und das boolesche Ausgangssignal (Überlauf bzw. Erreichen des Grenzwerts) werden **nur dann** an die Ausgänge weitergegeben, wenn ein Ereignis des Zählers (Zählimpuls oder Reset) ausgelöst wird. Dadurch wird vermieden, dass die Ausgänge bei jedem Systemtakt unnötig aktualisiert werden.

Verwendete Funktionsbausteine (FBs)

Im Subapplikationsnetzwerk sind folgende Bausteine enthalten:

- **DigitalOutput_Q1** (Typ: `logiBUS_QX`)
  - Parameter: `QI = TRUE`, `Output = Output_Q1`
- **DigitalInput_CLK_I1** (Typ: `logiBUS_IE`)
  - Parameter: `QI = TRUE`, `Input = Input_I1`, `InputEvent = BUTTON_SINGLE_CLICK`
- **DigitalInput_CLK_I2** (Typ: `logiBUS_IE`)
  - Parameter: `QI = TRUE`, `Input = Input_I2`, `InputEvent = BUTTON_SINGLE_CLICK`
- **E_CTU** (Typ: `E_CTU`)
  - Parameter: `PV = UINT#5`
- **E_SPLIT_4** (Typ: `E_SPLIT_4`)
  - Keine Parameter
- **E_MERGE_4** (Typ: `E_MERGE_4`)
  - Keine Parameter
- **E_D_FF_ANY** (Typ: `E_D_FF_ANY`)
  - Keine Parameter
- **E_D_FF** (Typ: `E_D_FF`)
  - Keine Parameter
- **Q_NumericValue** (Typ: `Q_NumericValue`)
  - Parameter: `u16ObjId = OutputNumber_N1`

Programmablauf und Verbindungen

1. **Eingänge**

- Taster **I1** (Single-Click) ist mit `DigitalInput_CLK_I1` verbunden und erzeugt ein Ereignis am Ausgang `IND`.
- Taster **I2** (Single-Click) ist mit `DigitalInput_CLK_I2` verbunden und erzeugt ebenfalls ein Ereignis an `IND`.

2. **Zähler E_CTU**
   - Der Ereignisausgang von `DigitalInput_CLK_I1` wird auf den Eingang `CU` (Count Up) des Zählers geführt.
   - Der Ereignisausgang von `DigitalInput_CLK_I2` wird auf den Eingang `R` (Reset) des Zählers geführt.
   - Der Zähler zählt bei jedem Impuls auf I1 den aktuellen Wert hoch. Bei Erreichen des Grenzwerts `PV = 5` wird der Ausgang `Q = TRUE`. Ein Reset setzt den Zählerstand auf `0` und `Q` auf FALSE.
   - Die Ereignisausgänge `CUO` (nach erfolgreichem Zählen) und `RO` (nach Reset) werden jeweils ausgelöst.
3. **Event-Bremsen-Logik**
   - Beide Ereignisse (`CUO` und `RO`) werden über den Baustein **E_SPLIT_4** auf vier Ausgänge verteilt.
   - Die vier Ausgänge werden im **E_MERGE_4** wieder zu einem gemeinsamen Ereignis zusammengeführt. Jeder eingehende Impuls erzeugt so ein einziges Ereignis am Ausgang `EO`.
   - Dieses zusammengeführte Ereignis dient als Takt (`CLK`) für die beiden Flip-Flops:
     - **E_D_FF_ANY** speichert den aktuellen Zählerstand (`CV`) und gibt ihn bei jedem Takt aus.
     - **E_D_FF** speichert den booleschen Ausgang (`Q`) des Zählers und gibt ihn ebenfalls bei jedem Takt aus.
4. **Ausgänge**
   - Der gespeicherte Zählerwert von `E_D_FF_ANY` wird über den Baustein **Q_NumericValue** als numerischer Wert auf der Anzeige `OutputNumber_N1` ausgegeben.
   - Der gespeicherte boolesche Wert von `E_D_FF` wird über **DigitalOutput_Q1** auf den digitalen Ausgang `Output_Q1` geschaltet.

Durch diese Verkettung wird sichergestellt, dass die Ausgänge nur dann aktualisiert werden, wenn der Zähler tatsächlich einen Schritt ausführt (Count oder Reset) – nicht bei jedem Systemtakt. Dies entspricht einer **Event-gesteuerten Bremse**.

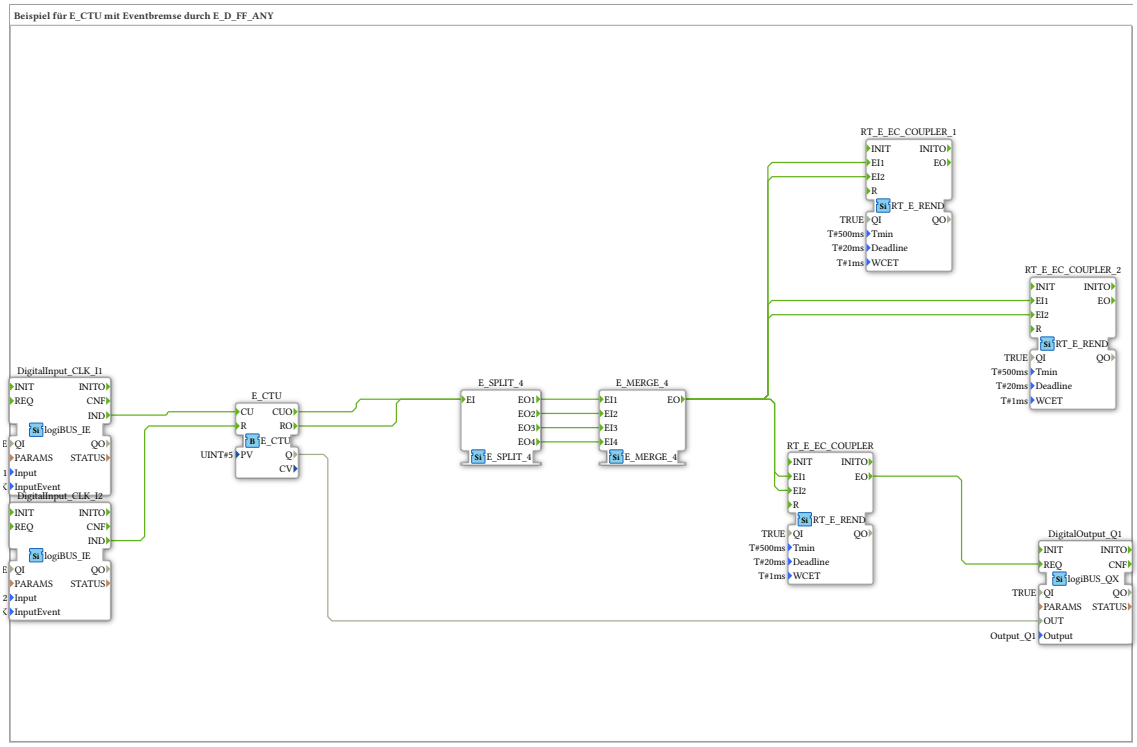
Zusammenfassung

Die Übung veranschaulicht die Kopplung von Ereignis- und Datenflüssen in einer 4diac-IDE Subapplikation. Der Aufwärtszähler wird durch zwei Taster gesteuert. Seine aktuellen Werte (Zählerstand und Überlaufsignal) werden über eine Kombination aus Ereignis-Splitter und -Merger auf Flip-Flops gegeben, die als Speicher dienen. Die Flip-Flops geben die gespeicherten Daten nur dann weiter, wenn ein Zähl- oder Reset-Ereignis vorliegt. Dies verhindert unnötige Aktualisierungen und zeigt, wie mit einfachen Mitteln eine **Event-Bremse** realisiert werden kann.

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

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

2.2.1.287 Uebung_080e: Beispiel für E_CTU mit Eventbremse durch E_D_FF_ANY



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert den Einsatz eines Aufwärtszählers (`E_CTU`) in Verbindung mit einer Ereignisbremse, realisiert durch `RT_E_REND`-Bausteine. Der Zähler wird über zwei Taster (Tipptaster mit Einfachklick) gesteuert: ein Taster zum Zählen (CU), ein weiterer zum Rücksetzen (R). Das Zählergebnis wird auf einen digitalen Ausgang gegeben. Die Ereignisbremse sorgt für eine zeitliche Entprellung und Entkopplung der Ereignisse. Es werden keine Sub-App-Bausteine verwendet; alle Funktionsbausteine sind Standard- oder gerätespezifische Bibliothekselemente.

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_CLK_I1**
  - **Typ**: `logiBUS::io::DI::logiBUS_IE`
  - **Parameter**:
    - `QI` = `TRUE`
    - `Input` = `Input_I1`
    - `InputEvent` = `BUTTON_SINGLE_CLICK`
  - **Funktion**: Erzeugt bei Tastendruck (Einfachklick) ein Ereignis am Ausgang `IND`. Dient als Zähltakt für den Aufwärtszähler.
- **DigitalInput_CLK_I2**
  - **Typ**: `logiBUS::io::DI::logiBUS_IE`
  - **Parameter**:
    - `QI` = `TRUE`
    - `Input` = `Input_I2`
    - `InputEvent` = `BUTTON_SINGLE_CLICK`
  - **Funktion**: Erzeugt bei Tastendruck ein Ereignis am Ausgang `IND`. Dient als Rücksetzsignal für den Zähler.
- **E_CTU** (Aufwärtszähler)
  - **Typ**: `iec61499::events::E_CTU`
  - **Parameter**:

    - PV = UINT#5
  - **Ereigniseingänge/-ausgänge**:
    - CU (zählen)
    - R (rücksetzen)
    - CUO (Ausgang nach Zählereignis)
    - RO (Ausgang nach Rücksetzen)
  - **Datenausgang**:
    - Q (aktueller Zählerstand, als UINT)
  - **Funktion**: Zählt bei jedem Ereignis an CU hoch (beginnend bei 0), bis der Wert PV (hier 5) erreicht ist; dann wird Q gesetzt. Bei Ereignis an R wird der Zähler zurückgesetzt.
- **E_SPLIT_4**
  - **Typ**: iec61499::events::E_SPLIT_4
  - **Funktion**: Verteilt ein eingehendes Ereignis (an EI) auf bis zu vier Ausgänge (EO1–EO4). Genutzt zur parallelen Verarbeitung nach Zählerereignissen.
- **E_MERGE_4**
  - **Typ**: iec61499::events::E_MERGE_4
  - **Funktion**: Führt bis zu vier eingehende Ereignisse (an EI1–EI4) zu einem gemeinsamen Ausgang (EO) zusammen. Fasst die parallelen Zweige wieder zu einem Ereignisstrom zusammen.
- **RT_E_EC_COUPLER** (drei Exemplare)
  - **Typ**: eclipse4diac::rtevents::RT_E_REND
  - **Parameter** (für alle drei):
    - QI = TRUE
    - Tmin = T#500ms
    - Deadline = T#20ms
    - WCET = T#1ms
  - **Ereigniseingänge**: EI1, EI2
  - **Ereignisausgang**: EO
  - **Funktion**: Stellt eine zeitliche Entkopplung und Mindestabstand von 500 ms zwischen Ereignissen sicher. Dient als „Ereignisbremse“, um schnelle Tippfolgen zu glätten.
- **DigitalOutput_Q1**
  - **Typ**: logiBUS::io::DQ::logiBUS_QX
  - **Parameter**:
    - QI = TRUE
    - Output = Output_Q1
  - **Ereigniseingang**: REQ
  - **Dateneingang**: OUT (vom Zählerstand Q des E_CTU)
  - **Funktion**: Schaltet den digitalen Ausgang Q1 auf den Wert, der am Dateneingang anliegt, sobald ein Ereignis an REQ eintrifft.

Programmablauf und Verbindungen

1. **Eingangsereignisse**:
   - Taster an Input_I1 (Einfachklick) erzeugt ein Ereignis an DigitalInput_CLK_I1.IND.
   - Taster an Input_I2 (Einfachklick) erzeugt ein Ereignis an DigitalInput_CLK_I2.IND.
2. **Zählersteuerung**:
   - Das Ereignis von DigitalInput_CLK_I1.IND wird mit dem Eingang E_CTU.CU verbunden – zählt hoch.
   - Das Ereignis von DigitalInput_CLK_I2.IND wird mit dem Eingang E_CTU.R verbunden – setzt zurück.
3. **Ereignisverteilung und -zusammenführung**:
   - Die Ausgänge E_CTU.CUO und E_CTU.RO werden auf den gemeinsamen Eingang E_SPLIT_4.EI geschaltet (beide Ereignisse lösen denselben Split aus).
   - Die vier Ausgänge EO1–EO4 werden auf die vier Eingänge EI1–EI4 des E_MERGE_4 gelegt. Dadurch wird jedes Zähler- oder Rücksetzereignis vierfach durchgeschleift (hier redundant, um alle Ausgänge zu bedienen).

- Der Merge-Ausgang `EO` führt diese zu einem einzigen Ereignisstrom zusammen.

4. **Ereignisbremse (RT_E_REND)**:
   - Das zusammengeführte Ereignis wird auf die Eingänge `EI1` und `EI2` aller drei `RT_E_REND`-Bausteine gelegt.
   - Der Ausgang `EO` des ersten `RT_E_REND` löst den `REQ`-Eingang des Digitalausgabebausteins `DigitalOutput_Q1` aus.
   - Die anderen beiden `RT_E_REND` sind ebenfalls im Netzwerk vorhanden (ggf. vorbereitet für weitere Ausgänge oder Redundanz), werden aber im aktuellen Datenfluss nicht direkt mit einem nachfolgenden FB verbunden.
5. **Datenfluss**:
   - Der aktuelle Zählerstand `E_CTU.Q` ist direkt mit dem Dateneingang `DigitalOutput_Q1.OUT` verbunden. Bei jedem Ereignis an `REQ` wird dieser Wert auf den physischen Ausgang `Output_Q1` übernommen.

Lernziele dieser Übung:

- Verständnis des Aufwärtszählers `E_CTU` in IEC 61499.
- Einsatz von Ereignis-Split und -Merge zur Parallelverarbeitung.
- Nutzung von `RT_E_REND` als zeitliche Entprellung (Ereignisbremse) zur Stabilisierung der Signalverarbeitung.

Schwierigkeitsgrad: Fortgeschritten (Ereignissteuerung mit mehreren Bausteinen). **Erforderliche Vorkenntnisse**: Grundlagen der 4diac-IDE, Aufbau einfacher Steuerungen mit digitalen Ein-/Ausgängen. **Start der Übung**: SubApp in ein leeres Projekt einfügen, die Taster und den Ausgang entsprechend der Hardware (z. B. logiBUS) verschalten.

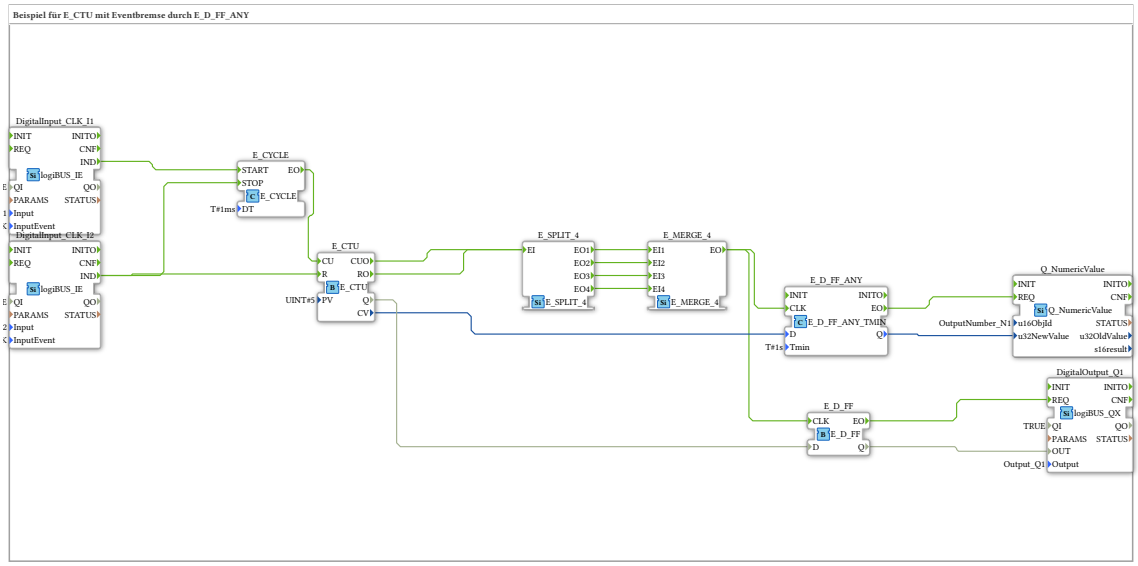
Zusammenfassung

Die Übung zeigt eine typische Zähleranwendung mit zwei Tastern, bei der ein Aufwärtszähler durch Tipptaster erhöht und zurückgesetzt wird. Der Zählerstand wird auf einen digitalen Ausgang gegeben. Durch die Verwendung von `E_SPLIT_4`, `E_MERGE_4` und vor allem der `RT_E_REND`-Bausteine wird das Ereignisverarbeitung robust gegenüber schnellen Tippfolgen gemacht – die Ereignisbremse erzwingt eine Mindestzeit von 500 ms zwischen zwei Verarbeitungsschritten. Dies verhindert ungewollte Mehrfachausgaben und entkoppelt die Eingabe von der Ausgabe.

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

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

2.2.1.288 Uebung_080e2: Beispiel für E_CTU mit Eventbremse durch E_D_FF_ANY



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung demonstriert die Verwendung eines Aufwärtszählers (E_CTU) in Kombination mit einer **Eventbremse**, realisiert durch den Baustein `E_D_FF_ANY_TMIN`. Der Zähler wird über einen zyklischen Ereignisgeber (E_CYCLE) inkrementiert, sobald ein Tastendruck an `DigitalInput_CLK_I1` erfolgt. Durch einen zweiten Tastendruck an `DigitalInput_CLK_I2` wird der Zähler zurückgesetzt und der Zyklus gestoppt. Die ausgegebenen Zählwerte werden nur dann an einen numerischen Ausgang weitergegeben, wenn die minimale Verweildauer (`Tmin`) des Signalzustands überschritten wird – dies verhindert ungewollte oder rauschende Werte. Ein zusätzlicher D-Flipflop-Baustein (`E_D_FF`) gibt den Zählerstatus (Q) als binäres Signal auf einen Digitalausgang.

Verwendete Funktionsbausteine (FBs)

Die Übung verwendet die folgenden vordefinierten Funktionsbausteine im Netzwerk:

Bausteinname	Typ	Parameter	Kurzbeschreibung
`DigitalInput_CLK_I1`	`logiBUS::io::DI::logiBUS_IO`	`QI = TRUE, Input = Input_I1, InputEvent = BUTTON_SINGLE_CLICK`	Erzeugt ein Ereignis (`IND`) bei einfachem Tastendruck auf Eingang I1.
`DigitalInput_CLK_I2`	`logiBUS::io::DI::logiBUS_IO`	`QI = TRUE, Input = Input_I2, InputEvent = BUTTON_SINGLE_CLICK`	Erzeugt ein Ereignis (`IND`) bei einfachem Tastendruck auf Eingang I2.
`E_CYCLE`	`iec61499::events::E_CYCLE`	`DT = T#1ms`	Zyklischer Ereignisgenerator; erzeugt nach Start alle 1 ms ein Ereignis (`EO`) bis zum Stopp.
`E_CTU`	`iec61499::events::E_CTU`	`PV = UINT#5`	Aufwärtszähler: zählt bei jedem Ereignis an `CU` hoch; gibt den aktuellen Zählwert (`CV`) und ein Überlaufsignal (`Q`) aus. Reset über `R`.
`E_SPLIT_4`	`iec61499::events::E_SPLIT_4`	–	Verteilt ein eingehendes Ereignis auf vier parallele Ausgänge (`EO1` … `EO4`).
`E_MERGE_4`	`iec61499::events::E_MERGE_4`	–	Fasst bis zu vier Eingangsereignisse (`EI1` … `EI4`) zu einem einzigen Ausgangsereignis (`EO`) zusammen.
`E_D_FF_ANY`	`iec61499::events::E_D_FF_ANY_TMIN`	`Tmin = T#1s`	D-Flipflop mit Mindestverweildauer: Übernimmt den Dateneingang `D` bei einem Ereignis an `CLK`, gibt den Zustand an `Q` aus, aber nur wenn das Ereignis mindestens `Tmin` lang anliegt.

E_D_FF	iec61499::events::E_D_FF	–	Standard-D-Flipflop: Übernimmt den Dateneingang D bei einem Ereignis an CLK und gibt den Zustand an Q aus.
DigitalOutput_Q1	logiBUS::io::DQ::logiBUS_DQ	QI = TRUE, Output = Output_Q1	Setzt den Digitalausgang Q1 auf den Wert des Eingangs OUT.
Q_NumericValue	isobus::UT::Q::Q_NumericValue	u16ObjId = OutputNumber_N1	Gibt einen numerischen Wert (32-Bit Integer) auf einer Visualisierungskomponente aus (hier: OutputNumber_N1).

Programmablauf und Verbindungen

Ereignisverbindungen

1. **Start des Zyklus**: Ein Tastendruck an DigitalInput_CLK_I1 löst das Ereignis IND aus. Dieses startet den E_CYCLE (über START).
2. **Zähler-Takt**: Der E_CYCLE erzeugt alle 1 ms ein Ereignis (EO), welches direkt an den Zähleingang CU des E_CTU geht.
3. **Zähler-Auswertung**: Der E_CTU gibt bei jedem Inkrement (oder Überlauf) ein Ereignis an CUO bzw. RO aus. Beide Ereignisse werden über E_SPLIT_4 auf vier parallele Pfade aufgeteilt.
4. **Zusammenführung**: Alle vier Ausgänge des E_SPLIT_4 werden im E_MERGE_4 wieder zu einem einzigen Ereignis vereinigt. Dadurch entsteht ein Ereignis bei jedem Zählerereignis (unabhängig von der Ursache).
5. **Eventbremse (E_D_FF_ANY)**: Das zusammengeführte Ereignis gelangt zum Takteingang CLK des E_D_FF_ANY. Dieses übernimmt den aktuellen Zählwert (CV des E_CTU) nur dann, wenn das Ereignis länger als 1 Sekunde stabil ansteht (Mindestverweildauer). Das Ausgangssignal Q des E_D_FF_ANY wird an den numerischen Ausgang Q_NumericValue weitergeleitet.
6. **Digitaler Ausgang (E_D_FF)**: Parallel dazu wird das gleiche Ereignis auch dem normalen E_D_FF zugeführt, das den binären Überlaufstatus (Q des E_CTU) speichert. Der Ausgang Q des E_D_FF steuert den Digitalausgang DigitalOutput_Q1.
7. **Stopp und Reset**: Ein Tastendruck an DigitalInput_CLK_I2 erzeugt ein Ereignis, das gleichzeitig den Zyklus (E_CYCLE.STOP) stoppt und den Zähler zurücksetzt (E_CTU.R).

Datenverbindungen

- E_CTU.CV (Aktueller Zählwert) ⟶ E_D_FF_ANY.D
- E_D_FF_ANY.Q ⟶ Q_NumericValue.u32NewValue (Ausgabe des gefilterten Zählwerts)
- E_CTU.Q (Überlauf/Status) ⟶ E_D_FF.D
- E_D_FF.Q ⟶ DigitalOutput_Q1.OUT (Binärer Ausgangszustand)

Hinweise zur Übung

- **Lernziele**: Verständnis für die Kombination von Aufwärtszählern, Ereignisflüssen und zeitverzögerter Wertübernahme (Eventbremse). Typische Anwendung: Entprellung von Zählimpulsen oder Glättung von Messwerten.
- **Schwierigkeitsgrad**: Fortgeschritten – Kenntnisse in IEC 61499 Ereignissteuerung und der Verwendung von E_D_FF_ANY werden vorausgesetzt.
- **Vorkenntnisse**: Grundlagen zu Funktionsbausteinen, Ereignisverbindungen und dem 4diac-IDE-Workflow.
- **Start der Übung**: Das SubApp-Objekt Uebung_080e2 muss in ein 4diac-Projekt eingebunden werden. Die Hardware-Eingänge (I1, I2) und Ausgänge (Q1, OutputNumber_N1) sind entsprechend der logiBUS-Konfiguration anzuschließen.

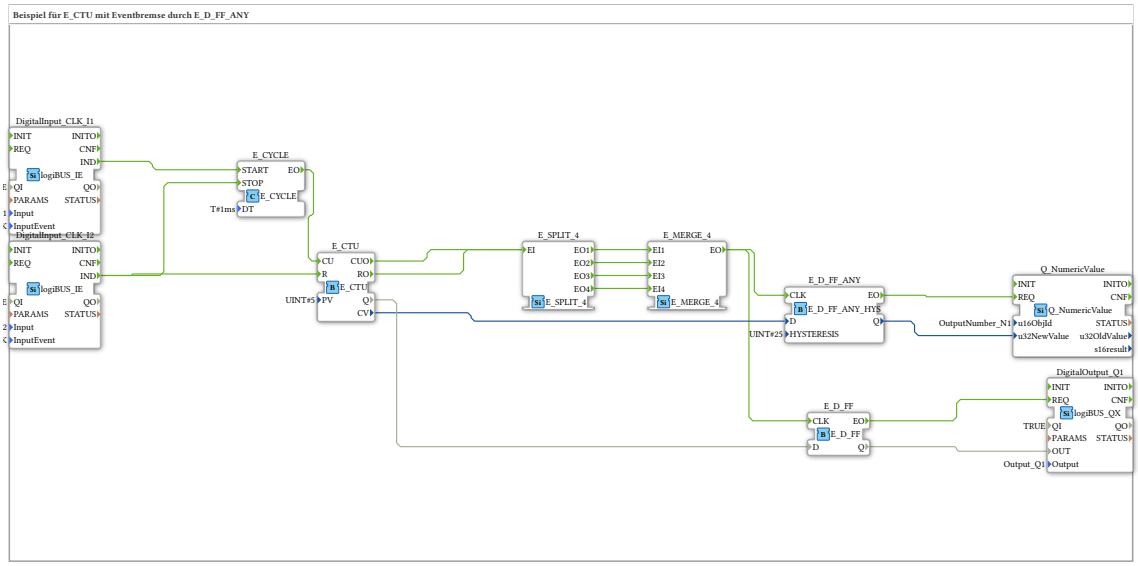
Zusammenfassung

Die Übung 080e2 veranschaulicht, wie ein Ereignisgesteuerter Zähler mit einer **temporalen Filterung** (Eventbremse) gekoppelt wird. Der Zähler wird durch einen Taster gestartet und durch einen zweiten Taster gestoppt bzw. zurückgesetzt. Der gefilterte Zählwert wird auf einem numerischen Display ausgegeben, während der binäre Überlaufstatus einen Digitalausgang ansteuert. Die Kombination von `E_SPLIT_4`, `E_MERGE_4` und `E_D_FF_ANY_TMIN` sorgt dafür, dass nur stabile Messwerte zur Anzeige gelangen. Dieses Muster eignet sich für robuste Zähleranwendungen in der Automatisierungstechnik.

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

- 🌐 E_CTU Event Counter Baustein auf ms-muc-docs.de
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de
- 🌐 IEC 61499 Events – Der Puls der Automatisierung auf ms-muc-docs.de

2.2.1.289 Uebung_080e3: Beispiel für E_CTU mit Eventbremse durch E_D_FF_ANY



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert den Einsatz des Funktionsbausteins **E_CTU** (Aufwärtszähler mit Ereignis-Steuerung) in Kombination mit einer **Eventbremse**, die durch einen **E_D_FF_ANY** (Flipflop mit Hysterese) realisiert wird. Durch das Zusammenspiel von zyklischen Zählimpulsen, manuellem Reset und einer Hysterese am Zählerwert wird ein Ausgangssignal nur dann geschaltet, wenn ein bestimmter Zählerstand erreicht und die Hysterese überschritten wird. Die Übung veranschaulicht die Verwendung von Ereignisverknüpfungen (E_SPLIT, E_MERGE) und die Ansteuerung einer digitalen Ausgabe sowie die Übergabe eines numerischen Werts an eine Ausgangsnummer.

Verwendete Funktionsbausteine (FBs)

Die Übung enthält keine weiteren Subapplikationen, sondern verwendet ausschließlich Basis-Funktionsbausteine aus den Bibliotheken `logiBUS`, `iec61499` und `isobus`. Nachfolgend werden die wichtigsten FBs beschrieben.

DigitalInput_CLK_I1 & DigitalInput_CLK_I2

- **Typ**: `logiBUS::io::DI::logiBUS_IE`
- **Parameter**:
  - `QI` = `TRUE`
  - `Input` = `Input_I1` bzw. `Input_I2`
  - `InputEvent` = `BUTTON_SINGLE_CLICK`
- **Funktion**: Erfasst einen Tastendruck (Einfachklick) an den physischen Eingängen I1/I2 und gibt bei jeder positiven Flanke ein Ereignis `IND` aus.

DigitalOutput_Q1

- **Typ**: `logiBUS::io::DQ::logiBUS_QX`
- **Parameter**:
  - `QI` = `TRUE`
  - `Output` = `Output_Q1`
- **Funktion**: Schaltet den digitalen Ausgang Q1 entsprechend dem anliegenden Datensignal `OUT`.

E_CYCLE

- **Typ**: `iec61499::events::E_CYCLE`
- **Parameter**:
  - `DT` = `T#1ms` (Periodendauer 1 ms)
- **Funktion**: Erzeugt nach dem Start (Ereigniseingang `START`) in regelmäßigen Abständen von 1 ms ein Ereignis am Ausgang `EO`. Kann durch das Ereignis `STOP` angehalten werden.

E_CTU

- **Typ**: `iec61499::events::E_CTU`
- **Parameter**:
  - `PV` = `UINT#5` (Vergleichswert 5)
- **Funktion**: Aufwärtszähler. Bei jedem Ereignis am Eingang `CU` wird der interne Zählerstand inkrementiert. Bei `R` (Reset) wird der Zähler auf 0 gesetzt. Am Ereignisausgang `CUO` wird nach jedem Zählimpuls ein Ereignis ausgegeben, am Ausgang `RO` bei Erreichen des Vergleichswerts `PV` (Zähler ≥ PV). Der aktuelle Zählerwert liegt am Datenausgang `CV` an, der boolesche Vergleichsstatus am Ausgang `Q`.

E_SPLIT_4

- **Typ**: `iec61499::events::E_SPLIT_4`
- **Funktion**: Verteilt ein eingehendes Ereignis (Eingang `EI`) auf vier parallele Ausgänge (`EO1`…`EO4`).

E_MERGE_4

- **Typ**: `iec61499::events::E_MERGE_4`
- **Funktion**: Führt bis zu vier eingehende Ereignisse (`EI1`…`EI4`) zu einem gemeinsamen Ereignisausgang `EO` zusammen (oder-Verknüpfung).

E_D_FF_ANY

- **Typ**: `logiBUS::signalprocessing::hysteresis::E_D_FF_ANY_HYS`
- **Parameter**:
  - `HYSTERESIS` = `UINT#25` (Hysteresebreite 25)
- **Funktion**: Ein Flipflop mit Hysterese. Bei einem Ereignis am Takteingang `CLK` wird der aktuelle Datenwert `D` (unsigned integer) übernommen. Der Ausgangswert `Q` wird nur dann aktualisiert, wenn die Differenz zum vorherigen Wert die Hysterese überschreitet. Dies verhindert ein ungewolltes Flackern bei geringen Änderungen. Der Ereignisausgang `EO` signalisiert eine Wertänderung.

E_D_FF

- **Typ**: `iec61499::events::E_D_FF`
- **Funktion**: Standard-D-Flipflop. Bei jedem Ereignis am Takteingang `CLK` wird der Dateneingang `D` (boolesch) übernommen und am Ausgang `Q` ausgegeben. Ein Ereignis am Ausgang `EO` zeigt die Übernahme an.

Q_NumericValue

- **Typ**: `isobus::UT::Q::Q_NumericValue`
- **Parameter**:
  - `u16ObjId` = `OutputNumber_N1`
- **Funktion**: Nimmt einen unsigned-integer-Wert am Dateneingang `u32NewValue` entgegen und gibt diesen bei einem Ereignis am Eingang `REQ` an die systemweit definierte Ausgangsnummer `N1` weiter (z. B. zur Anzeige auf einem Panel).

Programmablauf und Verbindungen

Die Übung verfolgt folgenden Ablauf:

1. **Initialisierung**: Ein Tastendruck an **I1** erzeugt ein Ereignis `IND` vom Baustein `DigitalInput_CLK_I1`. Dieses startet den **E_CYCLE**, der nun kontinuierlich alle 1 ms ein Ereignis an seinem Ausgang `EO` erzeugt.
2. **Zählen**: Das periodische Ereignis von **E_CYCLE** wird an den Zähleingang `CU` des **E_CTU** geleitet. Der Zähler zählt bei jedem Impuls hoch. Bei jedem Zählschritt wird über den Ausgang `CUO` ein Ereignis ausgegeben, ebenso sobald der Zählerstand den Vergleichswert `PV` (5) erreicht oder überschreitet (Ausgang `RO`).
3. **Ereignisvervielfachung und -zusammenführung**: Beide Ereignisausgänge des Zählers (`CUO` und `RO`) werden über einen **E_SPLIT_4** auf vier parallele Kanäle aufgeteilt. Diese vier Kanäle werden anschließend über einen **E_MERGE_4** wieder zu einem einzigen Ereignisstrom zusammengeführt. Somit erzeugt jeder Zählimpuls und jeder PV-Überschreitungsimpuls genau ein Ereignis am Ausgang des MERGE.
4. **Hysterese-gesteuertes Flipflop**: Dieses zusammengeführte Ereignis wird einerseits an den Takteingang `CLK` des **E_D_FF_ANY** gelegt. Der Dateneingang `D` erhält den aktuellen Zählerwert `CV` des E_CTU. Der **E_D_FF_ANY** übernimmt diesen Wert nur dann, wenn sich der Wert um mindestens `HYSTERESIS` (25) geändert hat. Bei einer solchen signifikanten Änderung gibt er ein Ereignis am Ausgang `EO` aus und legt den geglätteten Wert an `Q` an.
5. **Wertausgabe**: Das Ereignis des **E_D_FF_ANY** wird an den Eingang `REQ` des **Q_NumericValue** weitergeleitet. Dieser übernimmt den geglätteten Zählerwert (`u32NewValue`) von `E_D_FF_ANY.Q` und stellt ihn an der Ausgangsnummer `N1` bereit (z. B. für eine numerische Anzeige).
6. **Digitalausgang parallel**: Das zusammengeführte Ereignis vom **E_MERGE_4** wird ebenfalls an den Takteingang `CLK` eines normalen **E_D_FF** gelegt. Der Dateneingang `D` erhält den booleschen Status `Q` des E_CTU („Zähler ≥ PV“). Somit wird bei jedem Zählimpuls der aktuelle Vergleichsstatus im Flipflop gespeichert und am Ausgang `Q` ausgegeben. Ein Ereignis am Ausgang `EO` des Flipflops steuert den **DigitalOutput_Q1** an, der den booleschen Wert auf den physischen Ausgang Q1 legt.
7. **Reset-Funktion**: Ein Tastendruck an **I2** (zweiter Eingang) erzeugt ein Ereignis `IND` des **DigitalInput_CLK_I2**. Dieses Ereignis wird einerseits an den Reset-Eingang `R` des E_CTU gelegt (Zähler wird auf 0 gesetzt) und andererseits an den `STOP`-Eingang des E_CYCLE, wodurch die zyklische Erzeugung von Taktimpulsen angehalten wird. Somit wird der gesamte Zählvorgang zurückgesetzt und gestoppt.

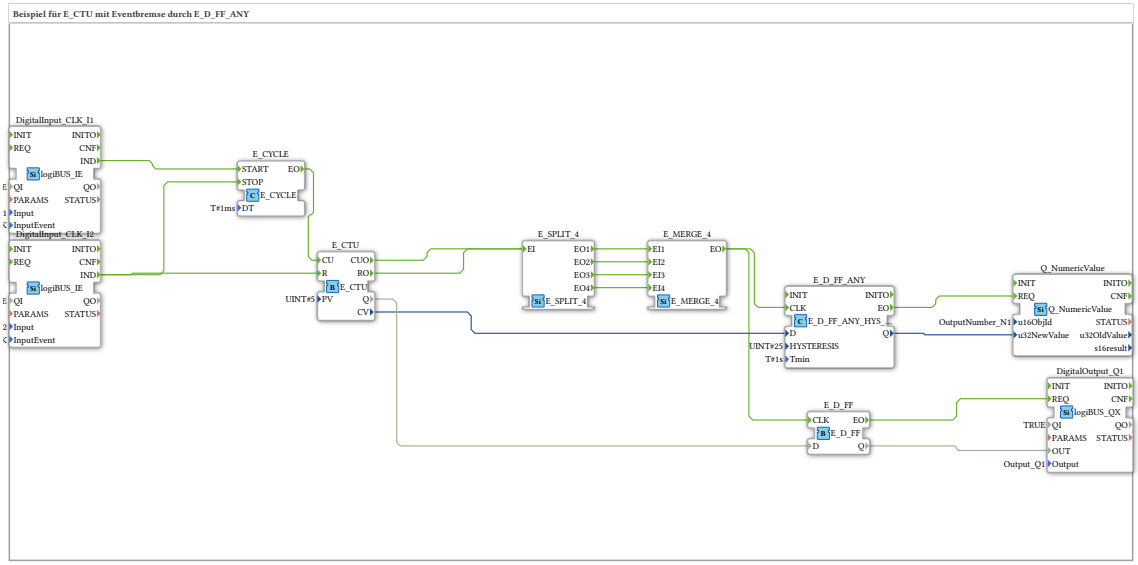
Zusammenfassung

Die Übung **Uebung_080e3** zeigt eine praxisnahe Kombination aus einem Ereignis-gesteuerten Zähler (**E_CTU**), einer Hysterese-basierten Wertglättung (**E_D_FF_ANY**) und einem booleschen Flipflop (**E_D_FF**). Durch das Zusammenspiel von **E_CYCLE**, **E_SPLIT_4** und **E_MERGE_4** wird ein robustes Ereignissystem aufgebaut, das den Zählerwert mit einer einstellbaren Hysterese überwacht und sowohl einen digitalen Ausgang als auch einen numerischen Wert ausgibt. Der Reset über einen zweiten Taster ermöglicht das Zurücksetzen und Stoppen der Zählung. Die Übung vermittelt grundlegende Konzepte der ereignisgesteuerten Steuerungstechnik, insbesondere die Verwendung von Ereignisverzweigungen, Zusammenführungen und die Bedeutung von Hysteresen bei der Signalverarbeitung.

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

- 🌐 E_CTU Event Counter Baustein auf ms-muc-docs.de

2.2.1.290 Uebung_080e4: Beispiel für E_CTU mit Eventbremse durch E_D_FF_ANY



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert den Einsatz eines **E_CTU** (Event-Zählers) in Kombination mit einer **Eventbremse**, realisiert durch einen **E_D_FF_ANY** (E_D_FlipFlop mit Hysterese und Minimalzeit). Ziel ist es, das Zählergebnis nur dann an eine numerische Ausgabe weiterzuleiten, wenn der Zählwert für eine bestimmte Zeit stabil ist. Dadurch werden Prellen oder kurzzeitige Änderungen unterdrückt.

Verwendete Funktionsbausteine (FBs)

Bausteinname	Typ	Parameter / Bemerkung
`DigitalInput_CLK_I1`	`logiBUS::io::DI::logiBUS_IE`	`Input = Input_I1`, `InputEvent = BUTTON_SINGLE_CLICK`
`DigitalInput_CLK_I2`	`logiBUS::io::DI::logiBUS_IE`	`Input = Input_I2`, `InputEvent = BUTTON_SINGLE_CLICK`
`E_CYCLE`	`iec61499::events::E_CYCLE`	`DT = T#1ms` (Taktgeber für Zählimpulse)
`E_CTU`	`iec61499::events::E_CTU`	`PV = UINT#5` (Zählschwelle)
`E_SPLIT_4`	`iec61499::events::E_SPLIT_4`	Verteilt ein Ereignis auf vier Ausgänge
`E_MERGE_4`	`iec61499::events::E_MERGE_4`	Sammelt Ereignisse von vier Eingängen zu einem Ausgang
`E_D_FF_ANY`	`logiBUS::signalprocessing::hysteresis::E_D_FF_ANY`	`HYSTERESIS = UINT#25`, `Tmin = T#1s` (Hysterese und Mindestzeit für stabilen Zustand)
`E_D_FF`	`iec61499::events::E_D_FF`	Standard-D-Flipflop für binäre Ausgabe
`Q_NumericValue`	`isobus::UT::Q::Q_NumericValue`	`u16ObjId = OutputNumber_N1` (Ausgabe eines numerischen Wertes)
`DigitalOutput_Q1`	`logiBUS::io::DQ::logiBUS_QX`	`Output = Output_Q1` (digitaler Ausgang)

Programmablauf und Verbindungen

Ereignis- und Datenfluss

1. **Zählimpulse generieren** Der Taktgeber `E_CYCLE` wird gestartet, sobald `DigitalInput_CLK_I1` ein Ereignis (`IND`) sendet. Stoppt wird er durch ein Ereignis von `DigitalInput_CLK_I2`. Der zyklische Ereignisausgang `EO` von `E_CYCLE` triggert den **Zähleingang `CU`** von `E_CTU`.
2. **Zähler rücksetzen** Ein Ereignis von `DigitalInput_CLK_I2` wird zusätzlich an den **Reset-Eingang `R`** von `E_CTU` geleitet.
3. **Zählerausgänge** Der Zähler gibt zwei Ereignisse aus:
   - `CUO` (Counter Overflow) – wird aktiv, wenn der Zählerstand `CV` den Parameter `PV` (hier 5) erreicht.
   - `RO` (Reset Overflow) – wird aktiv, wenn der Zähler zurückgesetzt wird und dabei den Bereich übersteigt (hier nicht relevant, aber beide Ereignisse werden verwendet).
4. **Ereignisverteilung und -zusammenführung** `CUO` und `RO` werden gemeinsam auf den Eingang `EI` von `E_SPLIT_4` geschaltet. `E_SPLIT_4` verteilt jedes ankommende Ereignis auf alle vier Ausgänge `EO1`…`EO4`. Diese vier Ausgänge sind mit den vier Eingängen `EI1`…`EI4` von `E_MERGE_4` verbunden. **Effekt:** Jedes Ereignis von `E_CTU` (egal ob `CUO` oder `RO`) wird sofort an den Ausgang `EO` von `E_MERGE_4` weitergegeben – es entsteht eine **logische ODER-Verknüpfung** der beiden Ereignisse.
5. **Eventbremse durch `E_D_FF_ANY`** Das zusammengeführte Ereignis speist den **Taktingang `CLK`** von `E_D_FF_ANY`. Dieser Baustein übernimmt den **Datenwert `D`** (den aktuellen Zählerstand `CV`) nur dann an den Ausgang `Q`, wenn der Wert für mindestens `Tmin = 1s` stabil bleibt (Hysterese von 25 Einheiten). Dadurch werden kurze Spitzen auf dem Zählerstand gefiltert.
6. **Numerische Ausgabe** Der stabile Zählerstand `Q` von `E_D_FF_ANY` wird über die Datenverbindung an den Eingang `u32NewValue` von `Q_NumericValue` übergeben. Das Ereignis `EO` von `E_D_FF_ANY` triggert die Ausgabe über den `REQ`-Eingang.
7. **Digitaler Ausgang** Parallel dazu wird das gleiche zusammengeführte Ereignis von `E_MERGE_4` auch an den **Taktingang `CLK`** eines normalen `E_D_FF` geleitet. Dieser übernimmt den **binären Datenwert `Q`** von `E_CTU` (den Zählerstatus: ob Schwelle erreicht) und gibt ihn über `EO` an `DigitalOutput_Q1` weiter. Der Ausgang `DigitalOutput_Q1` schaltet also immer dann ein, wenn der Zähler gerade seinen Endwert erreicht oder zurückgesetzt wird.

Lernziele

- Verständnis von **E_CTU (Event Counter)** und seinen Ereignisausgängen `CUO` und `RO`.
- Einsatz von **E_SPLIT_4** und **E_MERGE_4** zur Ereignissteuerung.
- Anwendung eines **E_D_FF_ANY mit Hysterese und Mindestzeit** zur Unterdrückung kurzzeitiger Änderungen (Eventbremse).
- Zusammenspiel von **numerischen und digitalen Ausgaben** basierend auf Zählerereignissen.

Schwierigkeitsgrad

Fortgeschritten – Grundkenntnisse in 4diac-IDE und IEC 61499 Ereignis-/Datenflüssen werden vorausgesetzt.

Benötigte Vorkenntnisse

- Grundlagen der 4diac-IDE: Erstellen von Subapplikationen, Verbinden von Bausteinen.
- Verständnis von Ereignis- und Datenkanten.
- Umgang mit logiBUS- bzw. isobus-Bibliotheken (bei Verwendung der Hardware-Simulation).

Starten der Übung

1. Importieren Sie die Subapplikation `Uebung_080e4` in Ihr 4diac-Projekt.
2. Stellen Sie sicher, dass die benötigten Bibliotheken (`logiBUS`, `iec61499`, `isobus`) verfügbar sind.
3. Weisen Sie die Ein-/Ausgänge `Input_I1`, `Input_I2`, `Output_Q1` und `OutputNumber_N1` passenden Hardware- oder Simulationsadressen zu.
4. Starten Sie die Ausführung und beobachten Sie das Verhalten beim Drücken der Taster (I1 Takt starten/stoppen, I2 Reset).

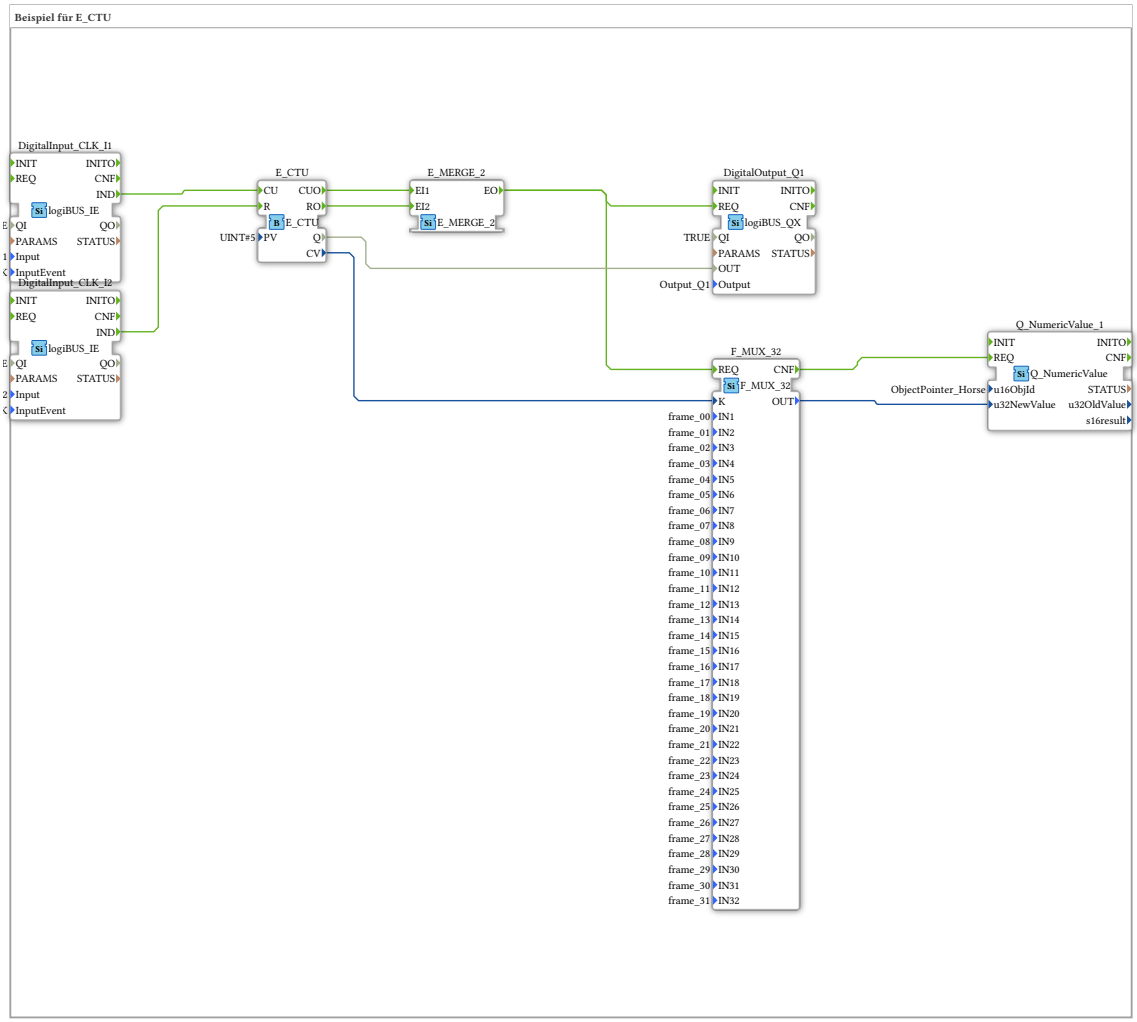
Zusammenfassung

Die Übung veranschaulicht, wie ein **E_CTU** mit Hilfe von **E_D_FF_ANY** und **E_D_FF** sowohl einen **stabilisierten Zählwert** als auch einen **sofortigen Binärstatus** ausgeben kann. Die Ereignisverarbeitung über Split/Merge sorgt dafür, dass sowohl Überlauf- als auch Rücksetzereignisse gleichberechtigt behandelt werden. Dies ist typisch für Anwendungen, bei denen ein Zählerwert erst nach einer gewissen Beruhigungszeit weiterverarbeitet werden soll (z. B. Entprellung von Sensordaten).

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

- 🌐 E_CTU Event Counter Baustein auf ms-muc-docs.de
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de
- 🌐 IEC 61499 Events – Der Puls der Automatisierung auf ms-muc-docs.de

2.2.1.291 Uebung_080f: Beispiel für E_CTU



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert die Verwendung des ereignisgesteuerten Aufwärtszählers `E_CTU` nach IEC 61499. Mit zwei Tastern wird der Zähler erhöht bzw. zurückgesetzt. Der aktuelle Zählerstand wird auf einem numerischen Display als animiertes Pferd (Einzelframes) dargestellt. Sobald der Zähler den vorgegebenen Grenzwert erreicht, wird ein digitaler Ausgang gesetzt.

Die Übung eignet sich für Anwender, die erste Schritte mit Zählern und Ereignisverkettungen in 4diac machen möchten.

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: Eingabe-Logik (Taster)

- **DigitalInput_CLK_I1 (Typ: `logiBUS::io::DI::logiBUS_IE`)**
  - Parameter:
    - `QI` = `TRUE`
    - `Input` = `Input_I1`
    - `InputEvent` = `BUTTON_SINGLE_CLICK`
  - Funktion: Stellt den ersten Taster (I1) als Ereignisquelle bereit. Jeder einfache Klick erzeugt ein Ereignis `IND`.
- **DigitalInput_CLK_I2 (Typ: `logiBUS::io::DI::logiBUS_IE`)**
  - Parameter:
    - `QI` = `TRUE`
    - `Input` = `Input_I2`
    - `InputEvent` = `BUTTON_SINGLE_CLICK`
  - Funktion: Stellt den zweiten Taster (I2) als Ereignisquelle bereit. Jeder einfache Klick erzeugt ein Ereignis `IND`.

Sub-Baustein: Aufwärtszähler

- **E_CTU (Typ: `iec61499::events::E_CTU`)**
  - Parameter:
    - `PV` = `UINT#5` (Grenzwert)
  - Funktion: Ein ereignisgesteuerter Aufwärtszähler. Jedes Ereignis am Eingang `CU` erhöht den internen Zähler CV um 1 und löst `CUO` aus. Ein Ereignis am Eingang `R` setzt den Zähler auf 0 zurück und löst `RO` aus. Der Ausgang `Q` wird `TRUE`, sobald `CV >= PV`.

Sub-Baustein: Ereigniszusammenführung

- **E_MERGE_2 (Typ: `iec61499::events::E_MERGE_2`)**
  - Funktion: Fasst zwei Ereigniseingänge (`EI1`, `EI2`) zu einem gemeinsamen Ereignisausgang `EO` zusammen. Sobald eines der beiden Ereignisse eintrifft, wird `EO` ausgelöst.

Sub-Baustein: Ausgabe-Logik (digitaler Ausgang)

- **DigitalOutput_Q1 (Typ: `logiBUS::io::DQ::logiBUS_QX`)**
  - Parameter:
    - `QI` = `TRUE`
    - `Output` = `Output_Q1`
  - Funktion: Schaltet den digitalen Ausgang Q1. Der Wert am Dateneingang `OUT` wird übernommen, wenn ein Ereignis an `REQ` eintrifft.

Sub-Baustein: Multiplexer

- **F_MUX_32 (Typ: `iec61131::selection::F_MUX_32`)**
  - Parameter:
    - `IN1` ... `IN32` = Konstanten `frame_00` ... `frame_31` (32 Einzelframes einer Pferdeanimation)
  - Funktion: Ein 32-fach Multiplexer. Je nach Wert am Auswahleingang `K` (0 ... 31) wird der entsprechende Dateingang `INx` an den Ausgang `OUT` durchgeschaltet.

Sub-Baustein: Numerische Anzeige

- **Q_NumericValue_1 (Typ: `isobus::UT::Q::Q_NumericValue`)**
  - Parameter:
    - `u16ObjId` = `ObjectPointer_Horse`
  - Funktion: Zeigt einen übergebenen 32-Bit-Wert (`u32NewValue`) numerisch an (hier: als animiertes Pferd über die Einzelframes). Ein Ereignis an `REQ` aktualisiert die Anzeige.

Programmablauf und Verbindungen

Der Ablauf wird durch Ereignisse gesteuert:

1. **Zählereingang**
   - Ein Klick auf Taster I1 erzeugt ein Ereignis `IND` am Baustein `DigitalInput_CLK_I1`.
   - Dieses Ereignis wird direkt an den Eingang `CU` von `E_CTU` geleitet. Der Zähler erhöht sich um 1.

2. **Reset**
   - Ein Klick auf Taster I2 erzeugt ein Ereignis `IND` an `DigitalInput_CLK_I2`.
   - Dieses Ereignis wird an den Eingang `R` von `E_CTU` geleitet. Der Zähler wird auf 0 zurückgesetzt.
3. **Ereigniszusammenführung**
   - Sowohl der Ausgang `CUO` (nach Zählererhöhung) als auch `RO` (nach Reset) von `E_CTU` werden an die Eingänge `EI1` und `EI2` des `E_MERGE_2` gelegt.
   - Der zusammengeführte Ausgang `EO` wird bei jeder Zähleränderung aktiv.
4. **Aktualisierung der Anzeige und des Ausgangs**
   - Das gemeinsame Ereignis `EO` wird parallel an zwei Bausteine weitergeleitet:
     - **Multiplexer**: Das Ereignis erreicht den `REQ`-Eingang von `F_MUX_32`. Der aktuelle Zählerstand `CV` (Datenverbindung von `E_CTU.CV` zu `F_MUX_32.K`) wählt das passende Frame aus. Der Multiplexer gibt das gewählte Frame an seinem Ausgang `OUT` aus.
     - **Digitale Anzeige**: Nachdem der Multiplexer fertig ist (`CNF`-Ereignis), wird das Ereignis an den `REQ`-Eingang von `Q_NumericValue_1` weitergegeben. Der Datenwert `OUT` des Multiplexers wird als neuer Anzeigewert übernommen.
   - Gleichzeitig wird das Ereignis `EO` auch an den `REQ`-Eingang von `DigitalOutput_Q1` gelegt. Der logische Wert `Q` von `E_CTU` (TRUE wenn `CV >= 5`) wird auf den Ausgang Q1 geschrieben.
5. **Kommentare im Netzwerk**
   - Ein Kommentar weist darauf hin, dass eine Typkonvertierung von `UINT` nach `UDINT` bei der Verbindung `CV` ⟶ `K` nicht notwendig ist, da `UDINT` immer einen `UINT` aufnehmen kann.
   - Ein weiterer Kommentar erklärt, dass der `E_MERGE_2` zwar weggelassen werden könnte, aber die Verwendung den Code sauberer hält (Vermeidung von sich kreuzenden Leitungen).

Zusammenfassung

Die Übung veranschaulicht den praktischen Einsatz eines ereignisgesteuerten Aufwärtszählers (`E_CTU`) in einer 4diac-Umgebung. Lernerfekt:

- Verständnis des Zusammenspiels von Ereignis- und Datenflüssen.
- Anwendung eines Zählers mit Ereignis-Reset.
- Nutzung eines Multiplexers zur Auswahl von Konstanten (Bildframes).
- Kombination von Hardware-Eingängen (Taster) und Ausgängen (digitaler Ausgang, Display).

Voraussetzungen: Grundkenntnisse der IEC 61499 Ereignisverarbeitung und der 4diac-IDE. Die Übung kann direkt in einem Simulationsprojekt oder auf echter logiBUS-Hardware ausgeführt werden.

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

- 🌐 E_CTU Event Counter Baustein auf ms-muc-docs.de
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de
- 🌐 IEC 61499 Events – Der Puls der Automatisierung auf ms-muc-docs.de

2.2.1.292 Uebung_081: Beispiel für E_CTD



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_081`. Hier wird das Prinzip des Rückwärtszählens bis zum Erreichen der Nullgrenze gezeigt.

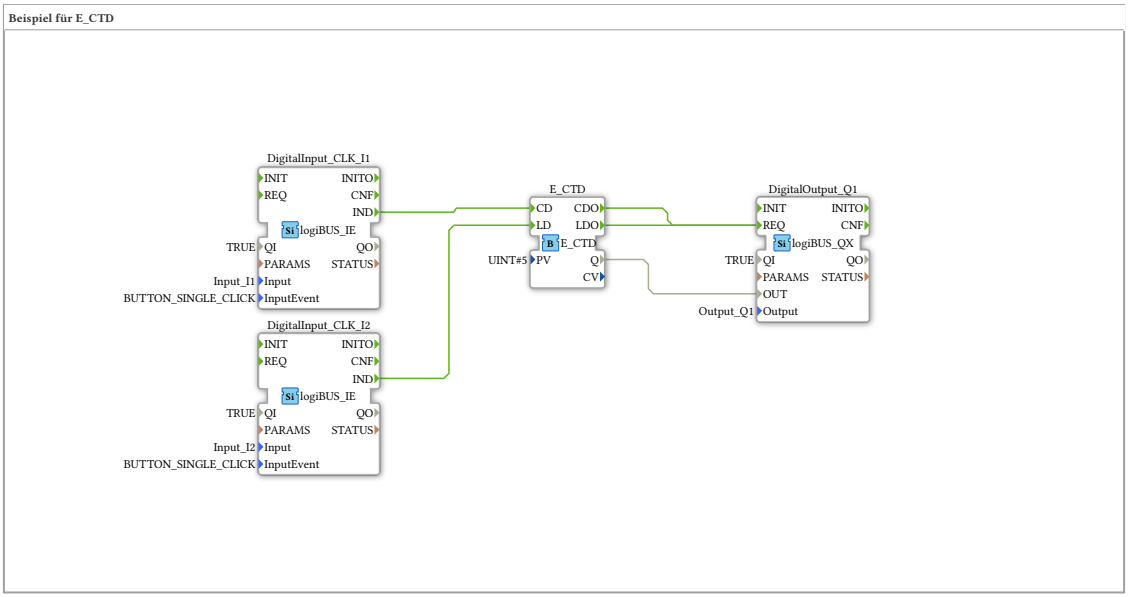
Ziel der Übung

Verwendung des Bausteins `E_CTD` (Event Count Down). Es wird demonstriert, wie ein Zähler mit einem Startwert geladen und durch Ereignisse bis auf Null dekrementiert wird.

Beschreibung und Komponenten

In `Uebung_081.SUB` wird ein Down-Counter zur Steuerung eines Ausgangs verwendet.

Funktionsbausteine (FBs)



- **`I1` (Count Down)**: Verringert den Zählerstand bei jedem Klick.
- **`I2` (Load)**: Lädt den Zähler mit dem Vorgabewert (`PV`).
- **`E_CTD`**: Der Zähler-Baustein. Der Parameter `PV` ist auf `5` eingestellt.
- **`DigitalOutput_Q1`**: Signalisiert das Erreichen der Nullgrenze.

Funktionsweise

1. **Laden**: Ein Klick auf **I2** triggert den Eingang `LD`. Der Zählerstand springt sofort auf
2. Der Ausgang `Q` wird `FALSE`.
3. **Zählen**: Jeder Klick auf **I1** (`CD`) verringert den Zählerstand (4, 3, 2, 1, 0).
4. **Grenzwert**: Sobald der Stand Null erreicht (`CV <= 0`), wechselt der Ausgang `Q` auf `TRUE`.
5. Die Lampe an **Q1** leuchtet auf.

Anwendungsbeispiel

Restmengen-Anzeige: In einem Saatgutbehälter befinden sich 5 Einheiten. Bei jeder Umdrehung der Dosierung wird ein Impuls (`CD`) ausgelöst. Sobald der Zähler bei Null ankommt, wird ein Alarm (`Q1`) ausgegeben, um den Fahrer zum Nachfüllen aufzufordern.

2.2.1.293 Uebung_082: Beispiel für E_CTUD



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_082`. Hier werden beide Zählrichtungen in einem Baustein kombiniert.

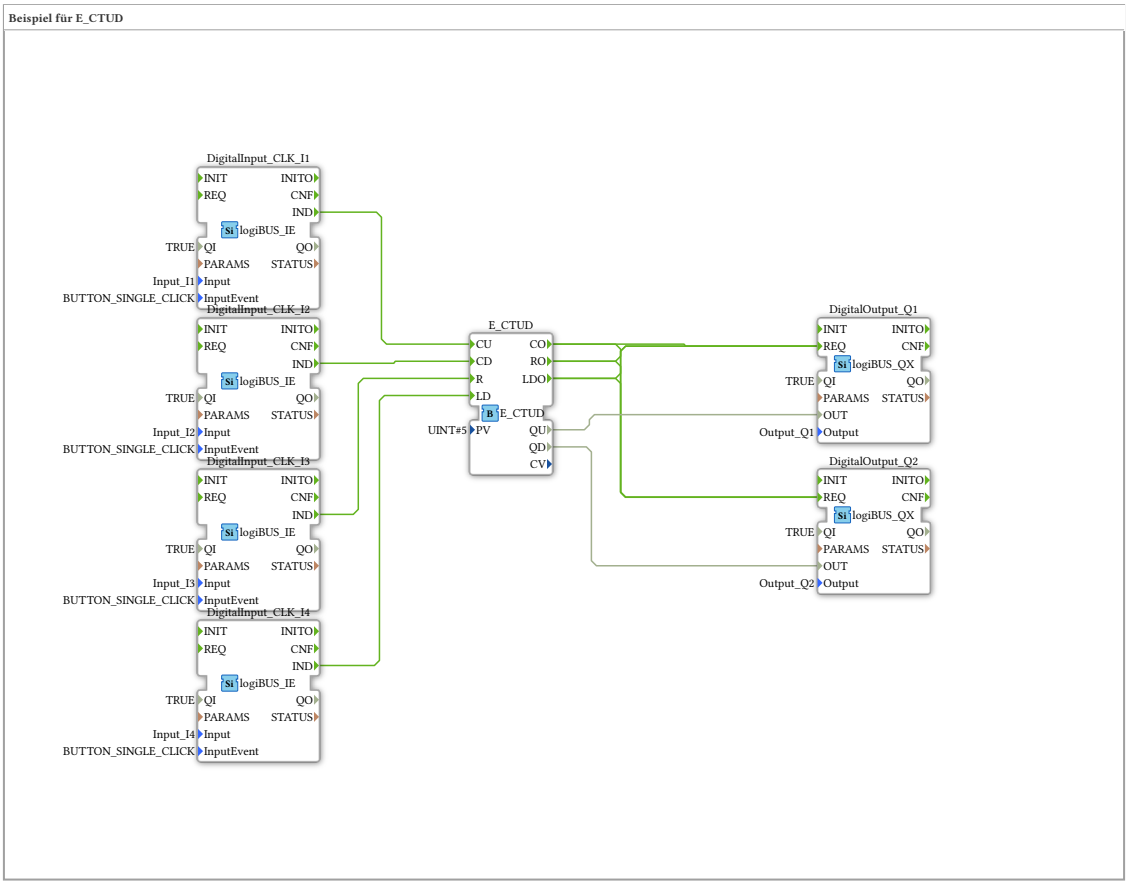
Ziel der Übung

Verwendung des Bausteins `E_CTUD` (Event Count Up/Down). Es wird gezeigt, wie man den Füllstand eines Speichers verwaltet, der sowohl Zu- als auch Abflüsse hat.

Beschreibung und Komponenten

Die Subapplikation `Uebung_082.SUB` nutzt vier Taster zur vollständigen Kontrolle des Zählers.

Funktionsbausteine (FBs)



- **`I1` (CU)**: Zählt aufwärts.
- **`I2` (CD)**: Zählt abwärts.
- **`I3` (R)**: Setzt den Zähler auf Null.
- **`I4` (LD)**: Lädt den Zähler mit dem Wert 5 (`PV`).
- **`Q1` (Upper Limit)**: Leuchtet, wenn der Zählerstand >= 5 ist.
- **`Q2` (Lower Limit)**: Leuchtet, wenn der Zählerstand <= 0 ist.

Funktionsweise

Der Baustein überwacht zwei Schwellwerte gleichzeitig:

- Der Ausgang `QU` reagiert auf die Obergrenze (`PV`).
- Der Ausgang `QD` reagiert auf die Untergrenze (Null).

Dies ermöglicht eine lückenlose Überwachung von Beständen oder Positionen innerhalb eines definierten Arbeitsbereichs.

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

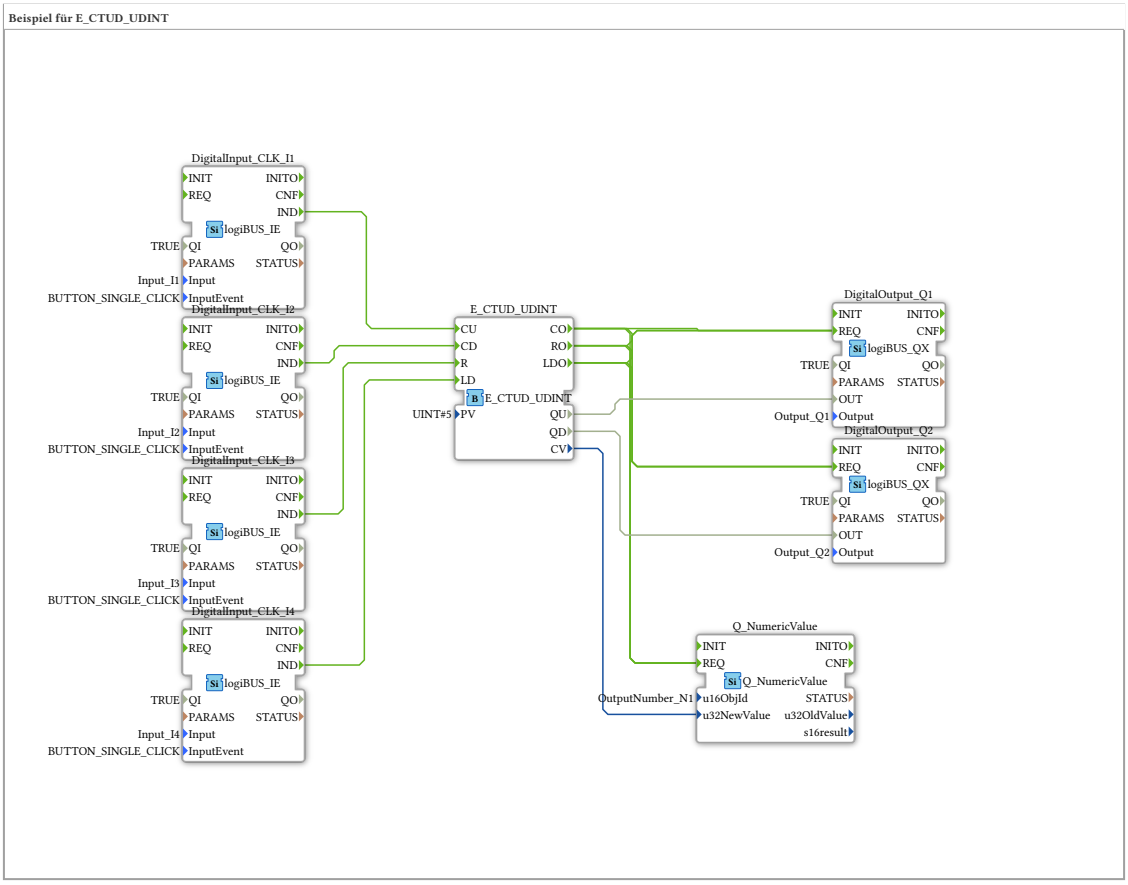
- 🌐 E_CTU Event Counter Baustein auf ms-muc-docs.de

2.2.1.294 Uebung_083: Beispiel für E_CTUD_UDINT



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_083`.

🎧 Podcast



- „Store Version“ – Dein Schlüssel zur Verwaltung von Objektdatenpools im nichtflüchtigen VT-Speicher (ISO 11783-6)
- ISO 11783-6: Softkeys und das Virtual Terminal verstehen – Dein Schlüssel zur Landmaschinen-Mechatronik
- ISOBUS Skalierung: Wenn der Ackerschlepper-Bildschirm nicht passt – Eine Einführung in ISO 11783-6
- ISOBUS-Balkendiagramm: Das Output Linear Bar Graph Objekt der ISO 11783-6 entschlüsselt
- ISOBUS-Bedienoberflächen: Wenn Tasten und Hauptanzeige unterschiedlich skalieren – ISO 11783-6 entschlüsselt

---

Übersicht

Diese Übung verwendet den Baustein `E_CTUD_UDINT`. Im Gegensatz zum Standard-Zähler (der meist nur bis 65.535 zählt) nutzt dieser Typ den Datentyp `UDINT` (Unsigned Double Integer). Damit können Ereignisse bis zu einem Wert von über 4 Milliarden gezählt werden. Zusätzlich zur Ansteuerung der Lampen `Q1` und `Q2` wird der aktuelle Zählerstand (`CV`) direkt an eine numerische Anzeige am ISOBUS-Terminal gesendet. Dies ermöglicht eine genaue Beobachtung des Zählvorgangs in Echtzeit.

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

- 🌐 E_CTU Event Counter Baustein auf ms-muc-docs.de

2.2.1.295 Uebung_084: Beispiel für E_CYCLE

NotebookLM

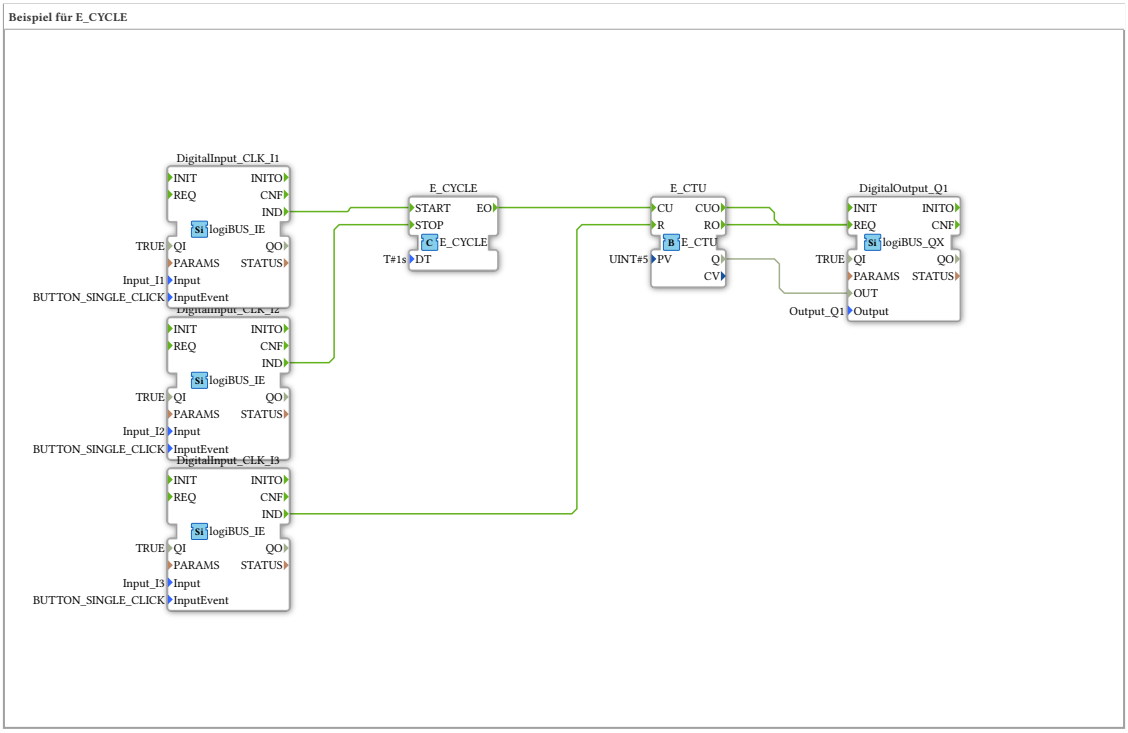
Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_084`. Hier wird der Zähler nicht manuell, sondern durch einen Taktgeber gesteuert.

Ziel der Übung

Kombination von Zeitbasis (`E_CYCLE`) und Ereignis-Zähler (`E_CTU`).

Funktionsweise

In `Uebung_084.SUB` wird der Zähler automatisch jede Sekunde inkrementiert.



- Taster **I1** startet den Taktgeber.
- Jedes Sekunde-Event vom `E_CYCLE` erreicht den `CU`-Eingang des Zählers.
- Nach 5 Sekunden erreicht der Zähler den Wert 5 und die Lampe `Q1` geht an.
- Taster **I2** stoppt den Taktgeber (Pause).
- Taster **I3** setzt den Zähler auf Null zurück.

Dies ist die Basis für die Implementierung von Zeit-Grenzwerten oder verzögerten Abschaltungen über längere Zeiträume.

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

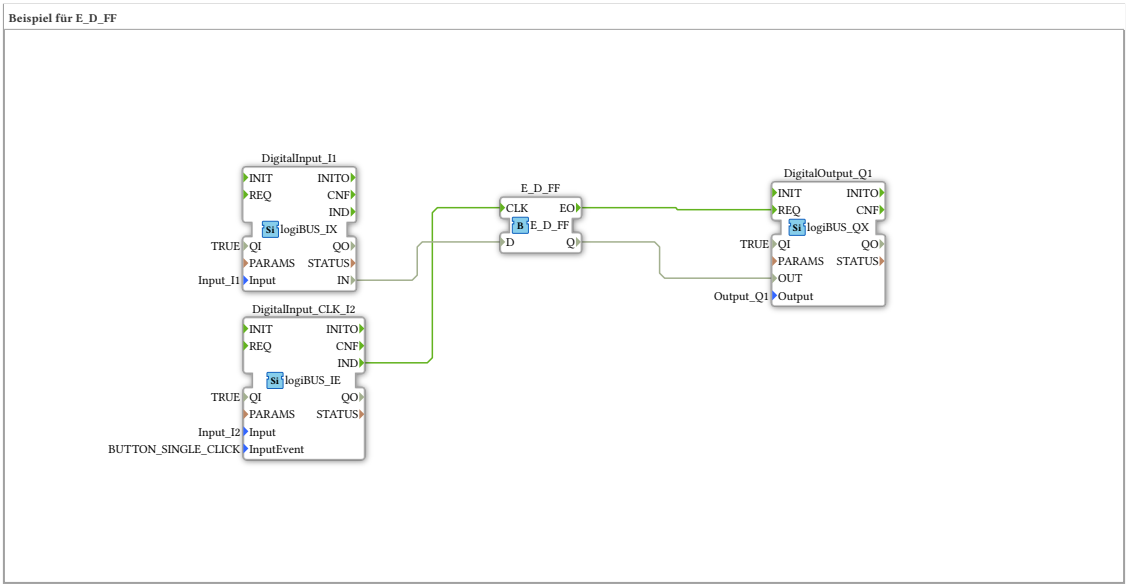
- 🌐 E_CTU Event Counter Baustein auf ms-muc-docs.de

2.2.1.296 Uebung_085: Beispiel für E_D_FF

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_085`. Hier wird das Prinzip des D-Flip-Flops (Delay- oder Data-FF) vorgestellt.

Podcast



- Das Relais im Detail: Schaltverstärker, Schutz und die Geheimnisse von A1/A2, 85/86 und der Hysterese

---

Ziel der Übung

Verwendung des Bausteins `E_D_FF`. Ziel ist es, einen Datenwert (TRUE/FALSE) erst in dem Moment zu übernehmen, in dem ein taktendendes Ereignis eintrifft.

Beschreibung und Komponenten

Die Subapplikation `Uebung_085.SUB` nutzt einen Daten-Eingang und einen Klick-Ereignis-Eingang.

Funktionsbausteine (FBs)

- **`I1` (Data)**: Liefert den Soll-Zustand.
- **`I2` (Clock)**: Liefert den Übernahme-Impuls.
- **`E_D_FF`**: Der Speicherbaustein. Er übernimmt den Wert am Eingang `D` nur dann an den Ausgang `Q`, wenn ein Ereignis am Eingang `CLK` empfängt.

Funktionsweise

Der Ausgang `Q1` folgt nicht sofort dem Schalter `I1`.

1. Der Nutzer stellt den Schalter `I1` auf TRUE. Am Ausgang passiert nichts.
2. Erst wenn der Nutzer zusätzlich auf Taster **I2** klickt, wird das `TRUE` in das Flip-Flop geladen und die Lampe geht an.
3. Wird `I1` wieder auf FALSE gestellt, bleibt die Lampe so lange an, bis erneut ein Klick auf **I2** erfolgt.

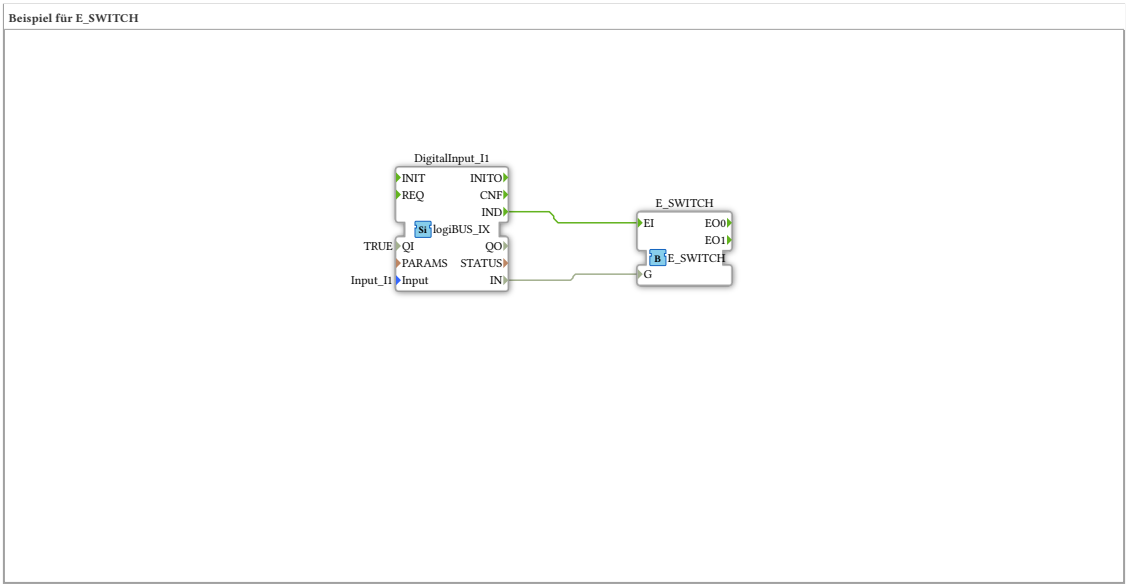
Dies ist eine fundamentale Methode zur zeitlichen Synchronisation von Signalen in der Digitaltechnik.

2.2.1.297 Uebung_086: Beispiel für E_SWITCH



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_086`.

📺 Video



- Der Katalog von 1863

🎧 Podcast

- Das Relais im Detail: Schaltverstärker, Schutz und die Geheimnisse von A1/A2, 85/86 und der Hysterese
- Das Technologie-Panorama von 1863: Lanz & Comp. und die Revolution der deutschen Landwirtschaft durch Import, Innovation und Guano

---

Übersicht

Verwendung des fundamentalen Bausteins `E_SWITCH`. In dieser Übung wird demonstriert, wie ein Ereignis-Strom (`EI`) basierend auf einem logischen Zustand (`G`) auf zwei verschiedene Pfade aufgeteilt wird.

- Ist der Schalter `I1` auf `FALSE`, landet das `IND`-Ereignis am Ausgang `EO0`.
- Ist der Schalter `I1` auf `TRUE`, landet das `IND`-Ereignis am Ausgang `EO1`. Dies ist die Basis für jede bedingte Programmausführung („If-Then-Else“) in der IEC 61499.

2.2.1.298 Uebung_087: Beispiel für E_DEMUX

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_087`. Hier wird die Auswahl eines Ereignispfads durch eine Kombination von Logikwerten demonstriert.

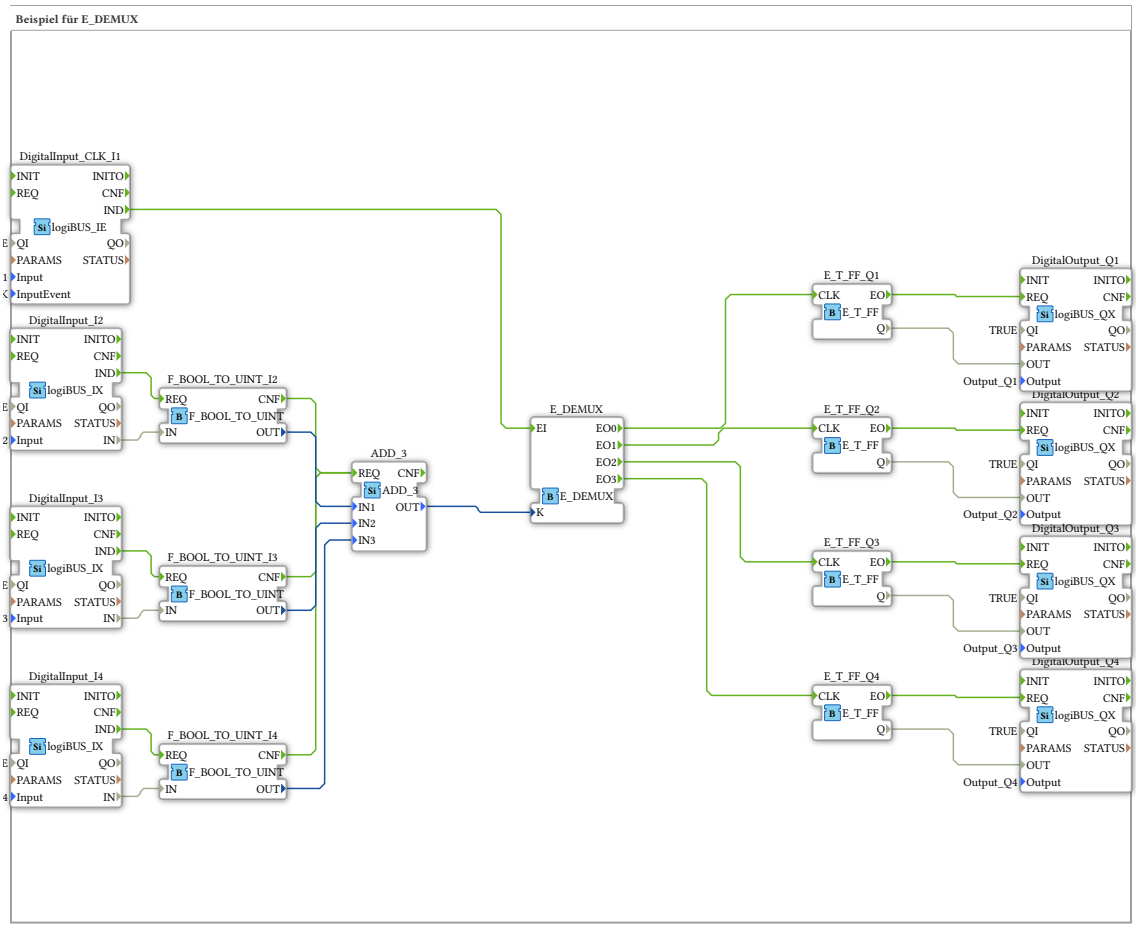
Ziel der Übung

Verwendung des `E_DEMUX` (Event Demultiplexer). Es wird gezeigt, wie ein zentrales „Ausführ-Ereignis“ (Klick auf Taster 1) an verschiedene Aktoren geleitet wird, wobei die Auswahl über die Kombination anderer Taster getroffen wird.

Beschreibung und Komponenten

Die Subapplikation `Uebung_087.SUB` nutzt eine Additions-Logik, um den Selektor-Eingang des Demultiplexers zu steuern.

Funktionsbausteine (FBs)



- **`I1` (Trigger)**: Das Ereignis, das verteilt werden soll.
- **`I2`, `I3`, `I4` (Selector)**: Bestimmen das Ziel.
- **`ADD_3`**: Summiert die Zustände der Taster 2, 3 und 4 auf.
- **`E_DEMUX`**: Leitet das Event von `I1` an den Ausgang weiter, dessen Nummer der berechneten Summe entspricht.

Funktionsweise

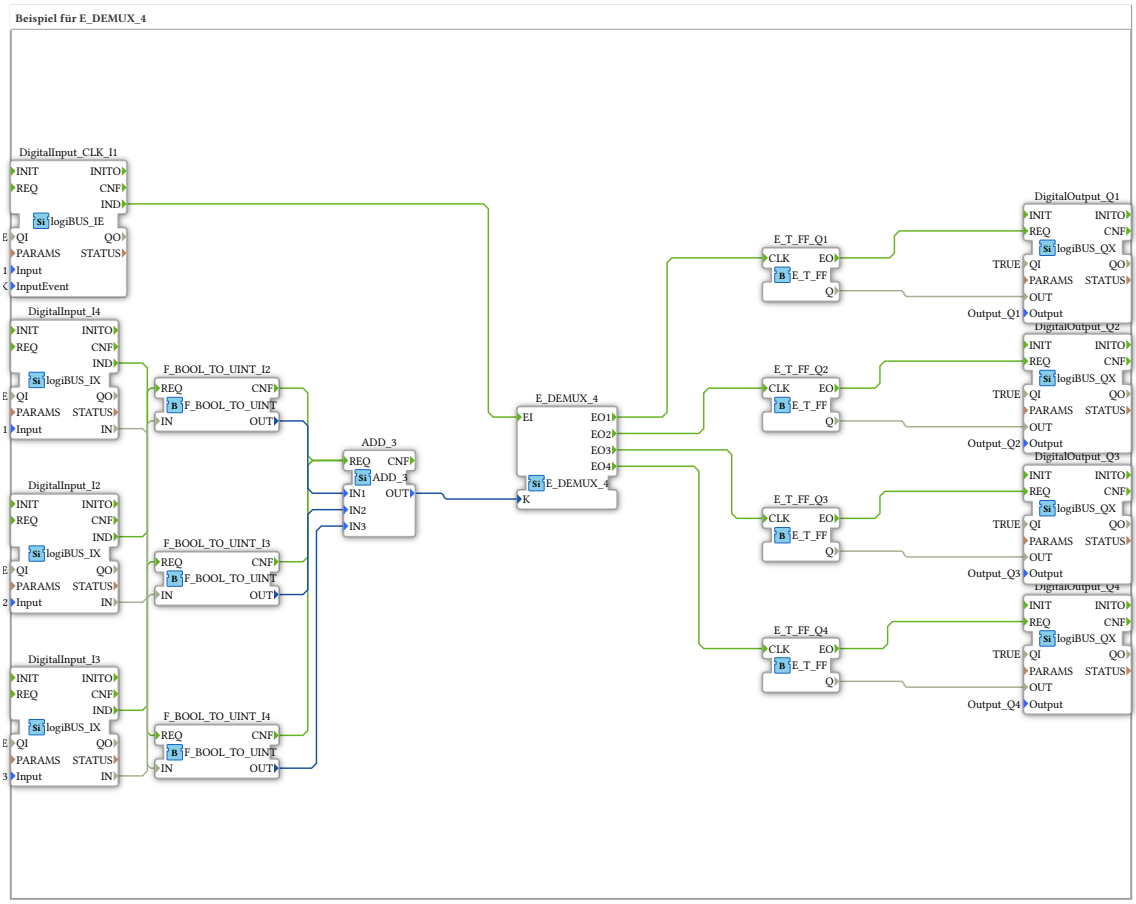
Die Anzahl der gedrückten „Wahl-Taster“ bestimmt, welche Lampe beim Klick auf **I1** toggelt:

- Kein Wahl-Taster gedrückt ➡️ Summe = 0 ➡️ Klick auf I1 toggelt **Q1**.
- Ein Wahl-Taster gedrückt ➡️ Summe = 1 ➡️ Klick auf I1 toggelt **Q2**.
- Zwei Wahl-Taster gedrückt ➡️ Summe = 2 ➡️ Klick auf I1 toggelt **Q3**.
- Alle drei Wahl-Taster gedrückt ➡️ Summe = 3 ➡️ Klick auf I1 toggelt **Q4**.

Anwendungsbeispiel

Indirekte Adressierung: Ein Bediener wählt über Kippschalter an seinem Bedienpult eine Gruppe von Düsen aus. Erst wenn er den zentralen Fußtaster (`I1`) betätigt, wird der Befehl an die gewählte Gruppe gesendet.

2.2.1.299 Uebung_087a1: Beispiel für E_DEMUX_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung demonstriert die Verwendung des E_DEMUX_4-Bausteins in einem Steuerungssystem. Das Programm zählt die Anzahl aktiver Eingänge und verteilt Ereignisse entsprechend auf verschiedene Ausgänge. Die Übung zeigt die Kombination von Ereignisverarbeitung und Datenverarbeitung in einem IEC 61499-System.

Verwendete Funktionsbausteine (FBs)

Hauptbausteine

- **E_DEMUX_4**: Ereignis-Demultiplexer mit 4 Ausgängen
- **ADD_3**: Addierer mit 3 Eingängen
- **E_T_FF**: T-Flipflop (4 Instanzen für Q1-Q4)
- **F_BOOL_TO_UINT**: Typkonvertierung von BOOL zu UINT (3 Instanzen)
- **logiBUS_IX**: Digitale Eingänge (4 Instanzen)
- **logiBUS_QX**: Digitale Ausgänge (4 Instanzen)

Sub-Bausteine

- **logiBUS_IX** (Digitale Eingänge)
  - **Typ**: Hardware-Eingangsbaustein
  - **Parameter**:
    - QI = TRUE (aktiviert)
    - Input = logiBUS_DI::Input_Ix (Hardware-Zuordnung)
    - InputEvent = logiBUS_DI_Events::BUTTON_SINGLE_CLICK (nur bei CLK_I1)
- **logiBUS_QX** (Digitale Ausgänge)

- **Typ**: Hardware-Ausgangsbaustein
- **Parameter**:
  - QI = TRUE (aktiviert)
  - Output = logiBUS_DO::Output_Qx (Hardware-Zuordnung)

Programmablauf und Verbindungen

Signalfluss

1. **Eingangsverarbeitung**:
   - Drei digitale Eingänge (I2, I3, I4) werden über F_BOOL_TO_UINT in UINT-Werte konvertiert
   - Ein spezieller Takt-Eingang (CLK_I1) mit Einzelklick-Erkennung
2. **Berechnung**:
   - Die drei UINT-Werte werden im ADD_3-Baustein summiert
   - Das Ergebnis bestimmt den Ausgangskanal des E_DEMUX_4
3. **Ereignisverteilung**:
   - E_DEMUX_4 verteilt das Takt-Ereignis auf einen von vier Ausgängen basierend auf der Summe
   - 0 aktive Tasten ⟶ Q1
   - 1 aktive Taste ⟶ Q2
   - 2 aktive Tasten ⟶ Q3
   - 3 aktive Tasten ⟶ Q4
4. **Ausgangssteuerung**:
   - Vier T-Flipflops (E_T_FF) schalten die entsprechenden Ausgänge (Q1-Q4) bei jedem Takt-Ereignis

Verbindungen

- **Ereignisverbindungen**: Verknüpfen IND-Ereignisse der Eingänge mit REQ-Ereignissen der Konverter und weiter zum ADD_3 und E_DEMUX_4
- **Datenverbindungen**: Übertragen die Eingangszustände durch die Konvertierung zur Addition und weiter zum Demultiplexer

Lernziele

- Verständnis des E_DEMUX_4-Bausteins
- Kombination von Ereignis- und Datenverarbeitung
- Umgang mit Hardware-Ein-/Ausgängen im logiBUS-System
- Implementierung von Zähl- und Verteilungslogik

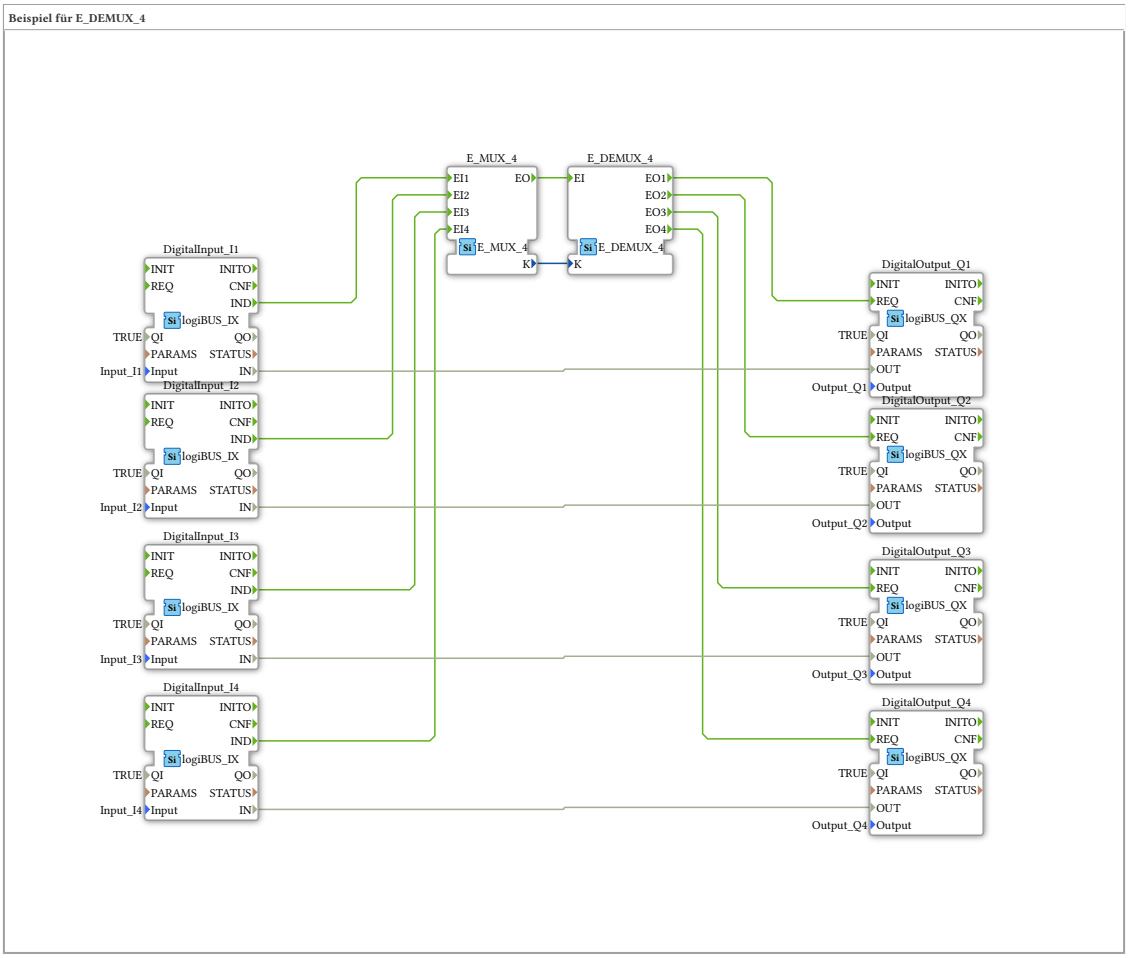
Schwierigkeitsgrad: Mittel

Vorkenntnisse: Grundlagen IEC 61499, Ereignisverarbeitung, Datenkonvertierung

Zusammenfassung

Diese Übung zeigt ein praktisches Beispiel für die Verwendung eines Ereignis-Demultiplexers in Kombination mit arithmetischen Operationen. Das System zählt aktivierte Eingänge und verteilt Takt-Ereignisse entsprechend auf verschiedene Ausgänge. Die Implementierung demonstriert effektiv die Verknüpfung von Hardware-Ein-/Ausgängen mit logischer Verarbeitung in einem IEC 61499-konformen Steuerungssystem.

2.2.1.300 Uebung_087a2: Beispiel für E_DEMUX_4



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung demonstriert die Funktionsweise des E_DEMUX_4 Bausteins in der 4diac-IDE. Die Anwendung zeigt, wie Ereignisse über einen Multiplexer und Demultiplexer verteilt werden können, um verschiedene digitale Ausgänge zu steuern.

Verwendete Funktionsbausteine (FBs)

Die Übung verwendet folgende Haupt-Funktionsbausteine:

- **E_MUX_4**: 4-fach Ereignis-Multiplexer
- **E_DEMUX_4**: 4-fach Ereignis-Demultiplexer
- **DigitalInput_I1-I4**: Digitale Eingänge (logiBUS_IX)
- **DigitalOutput_Q1-Q4**: Digitale Ausgänge (logiBUS_QX)

Programmablauf und Verbindungen

Ereignisverbindungen

- Die IND-Ereignisse der vier digitalen Eingänge (I1-I4) sind mit den entsprechenden Eingängen des E_MUX_4 Bausteins verbunden
- Der Ausgang EO des E_MUX_4 ist mit dem Eingang EI des E_DEMUX_4 verbunden
- Die vier Ausgänge des E_DEMUX_4 (EO1-EO4) sind mit den REQ-Eingängen der entsprechenden digitalen Ausgänge (Q1-Q4) verbunden

Datenverbindungen

- Der K-Ausgang des E_MUX_4 ist mit dem K-Eingang des E_DEMUX_4 verbunden

- Jeder digitale Eingang ist direkt mit seinem entsprechenden digitalen Ausgang verbunden (I1→Q1, I2→Q2, I3→Q3, I4→Q4)

Funktionsweise

Der E_MUX_4 Baustein sammelt Ereignisse von den vier digitalen Eingängen und leitet sie über einen gemeinsamen Ausgang weiter. Der E_DEMUX_4 Baustein verteilt diese Ereignisse basierend auf dem K-Wert an die entsprechenden digitalen Ausgänge. Die direkten Datenverbindungen zwischen Eingängen und Ausgängen sorgen für eine 1:1-Signalübertragung.

Lernziele

- Verständnis der Funktionsweise von Multiplexern und Demultiplexern
- Umgang mit Ereignis- und Datenverbindungen in 4diac
- Implementierung von Signalverteilungssystemen
- Anwendung der logiBUS-Schnittstellen für digitale Ein- und Ausgänge

Schwierigkeitsgrad

Mittel - Grundkenntnisse in 4diac und IEC 61499 werden vorausgesetzt

Vorkenntnisse

- Grundlagen der IEC 61499 Standard
- Kenntnisse der 4diac-IDE Oberfläche
- Verständnis von Ereignis- und Datenflüssen

Zusammenfassung

Diese Übung vermittelt praktische Erfahrungen mit Ereignis-Multiplexing und -Demultiplexing in 4diac. Sie zeigt, wie komplexe Signalverteilungen mit den Standard-Bausteinen E_MUX_4 und E_DEMUX_4 realisiert werden können. Die direkte Verbindung zwischen digitalen Ein- und Ausgängen demonstriert gleichzeitig die grundlegende Signalverarbeitung in Automatisierungssystemen.

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

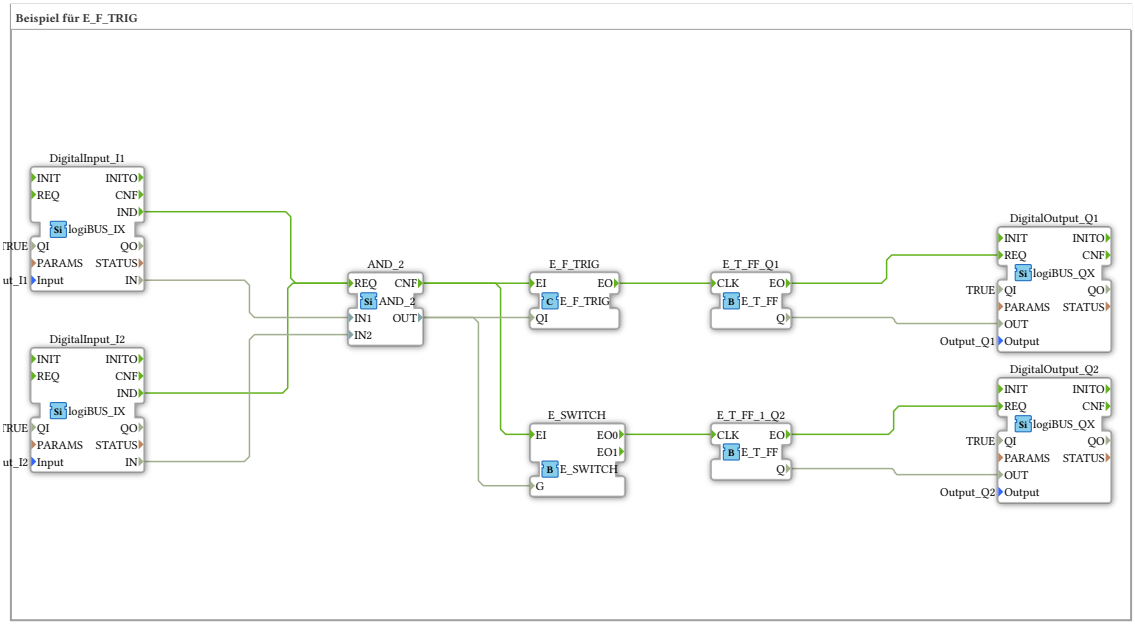
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.2.1.301 Uebung_088: Beispiel für E_F_TRIG



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_088`. Hier wird die gezielte Reaktion auf das Ende eines Signals (Ausschaltflanke) demonstriert.

🎧 Podcast



- Agrar-Revolution 1883: Wie Max Eyth Englands Landwirtschaft modernisierte

---

Ziel der Übung

Verwendung des Bausteins `E_F_TRIG` (Falling Edge Trigger). Im Gegensatz zum einfachen `E_SWITCH` filtert dieser Baustein alle Ereignisse heraus, außer den Moment des Übergangs von `TRUE` nach `FALSE`.

Beschreibung und Komponenten

In `Uebung_088.SUB` wird die Reaktion auf eine UND-Logik verglichen.

Funktionsweise

1. Zwei Taster `I1` und `I2` werden über ein `AND_2` Gatter verknüpft.
2. Das Ergebnis (`OUT`) liegt am Eingang `QI` des Flanken-Triggers an.
3. **Positive Flanke**: Schaltet man die Taster ein, passiert am Ausgang nichts.
4. **Negative Flanke**: Erst in dem Moment, in dem die UND-Bedingung wieder verloren geht (indem man **einen der beiden** Taster loslässt), feuert `E_F_TRIG.EO`.
5. Das Flip-Flop toggelt, die Lampe wechselt den Zustand.

Anwendungsbeispiel

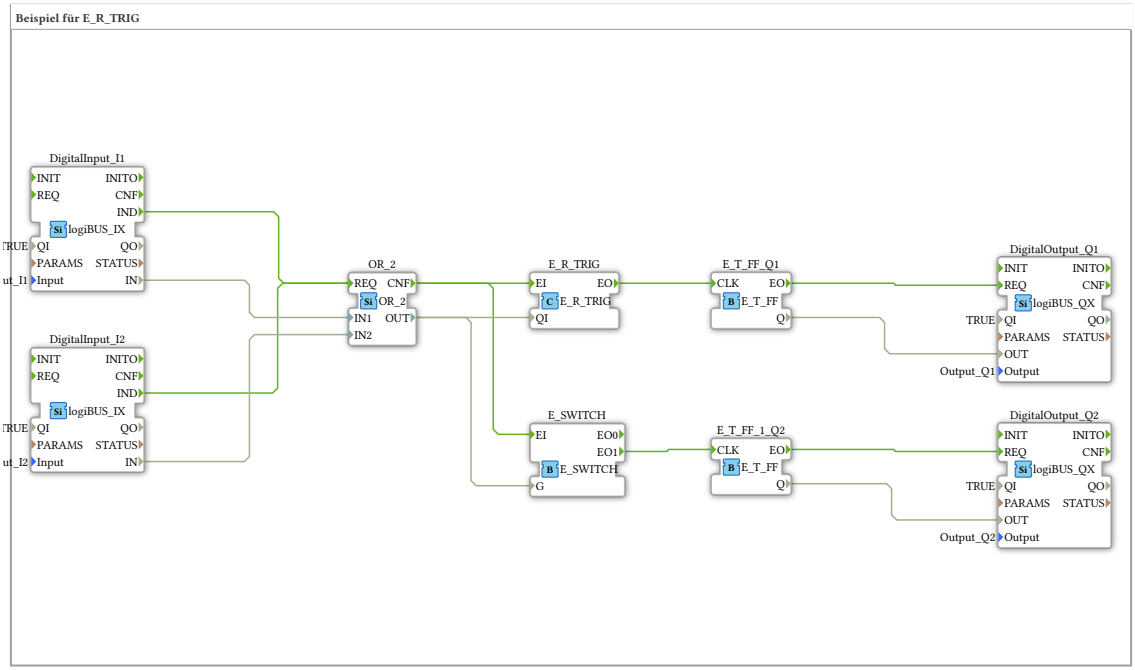
Sicherheits-Check beim Ausschalten: Eine Reinigungsfunktion soll erst dann starten, wenn der Hauptschalter der Maschine ausgeschaltet wurde. Der `F_TRIG` erkennt diesen Ausschalt-Moment und löst den Folgeschritt aus.

2.2.1.302 Uebung_089: Beispiel für E_R_TRIG

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_089`.

🎧 Podcast



- Apfelwein-Allzweckwaffe und Stickstoff-Revolution: Die Landwirtschaft Mittelfrankens 1892 im Zeitungs-Check

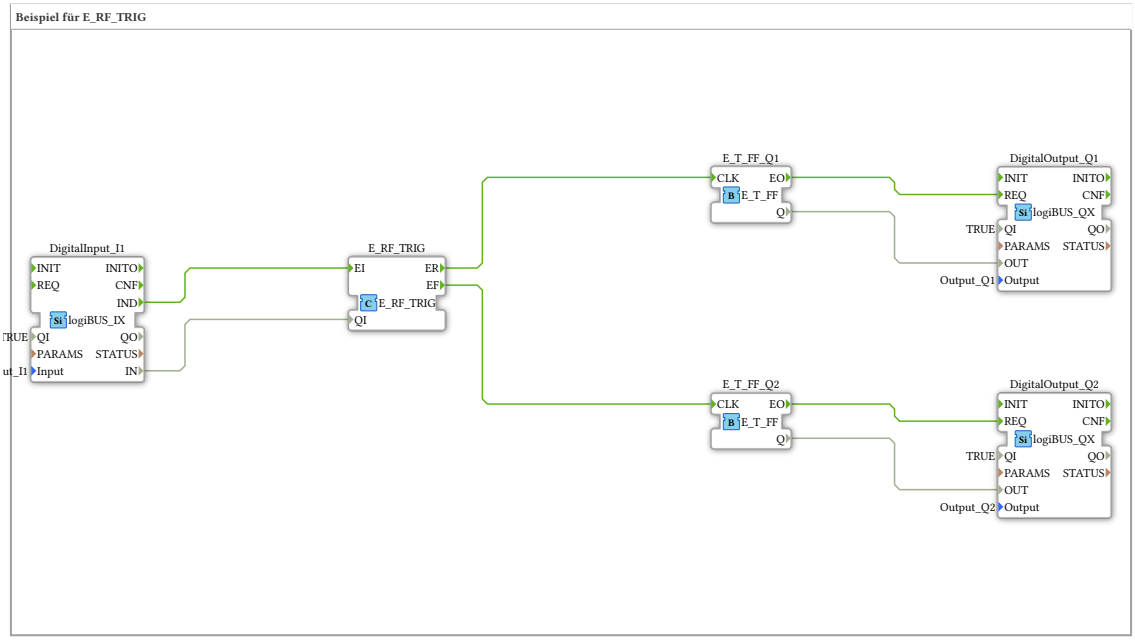
---

Übersicht

Pendant zur vorherigen Übung unter Verwendung des Bausteins `E_R_TRIG` Rising Edge Trigger. Das Flip-Flop wird hier genau in dem Moment getriggert, in dem eine ODER-Bedingung (`I1 OR I2`) wahr wird. Das bedeutet: Sobald man den **ersten** der beiden Taster drückt, toggelt die Lampe. Das Drücken des zweiten Tasters (während der erste noch gehalten wird) hat keine Auswirkung, da die Logik bereits auf TRUE steht und keine erneute steigende Flanke erzeugt wird.

2.2.1.303 Uebung_089a: Beispiel für E_RF_TRIG

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_089a (Beispiel für E_RF_TRIG).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Beispiel für E_RF_TRIG**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_089a.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IX.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
- **E_RF_TRIG**: Instanz des Typs iec61499::events::E_RF_TRIG.
- **E_T_FF_Q1**: Instanz des Typs iec61499::events::E_T_FF.
- **E_T_FF_Q2**: Instanz des Typs iec61499::events::E_T_FF.
- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QX.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
- **DigitalOutput_Q2**: Instanz des Typs logiBUS::io::DQ::logiBUS_QX.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q2

Verbindungen und Schnittstellen

Ereignisverbindungen:

- DigitalInput_I1.IND -> E_RF_TRIG.EI
- E_RF_TRIG.ER -> E_T_FF_Q1.CLK
- E_RF_TRIG.EF -> E_T_FF_Q2.CLK
- E_T_FF_Q1.EO -> DigitalOutput_Q1.REQ
- E_T_FF_Q2.EO -> DigitalOutput_Q2.REQ

Datenverbindungen:

- DigitalInput_I1.IN -> E_RF_TRIG.QI
- E_T_FF_Q1.Q -> DigitalOutput_Q1.OUT
- E_T_FF_Q2.Q -> DigitalOutput_Q2.OUT

Hinweise aus dem Modell

> Ein einziger Eingang, EIN Signal - E_RF_TRIG liefert beide Flanken gleichzeitig (ER=steigend, EF=fallend). Q1 togglet bei steigender, Q2 bei fallender Flanke.

Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_089a demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.2.1.304 Uebung_090a1: Beispiel für F_MUX_2



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_090a1`. Hier wird die Auswahl eines Datenwertes basierend auf einer Adresse demonstriert.

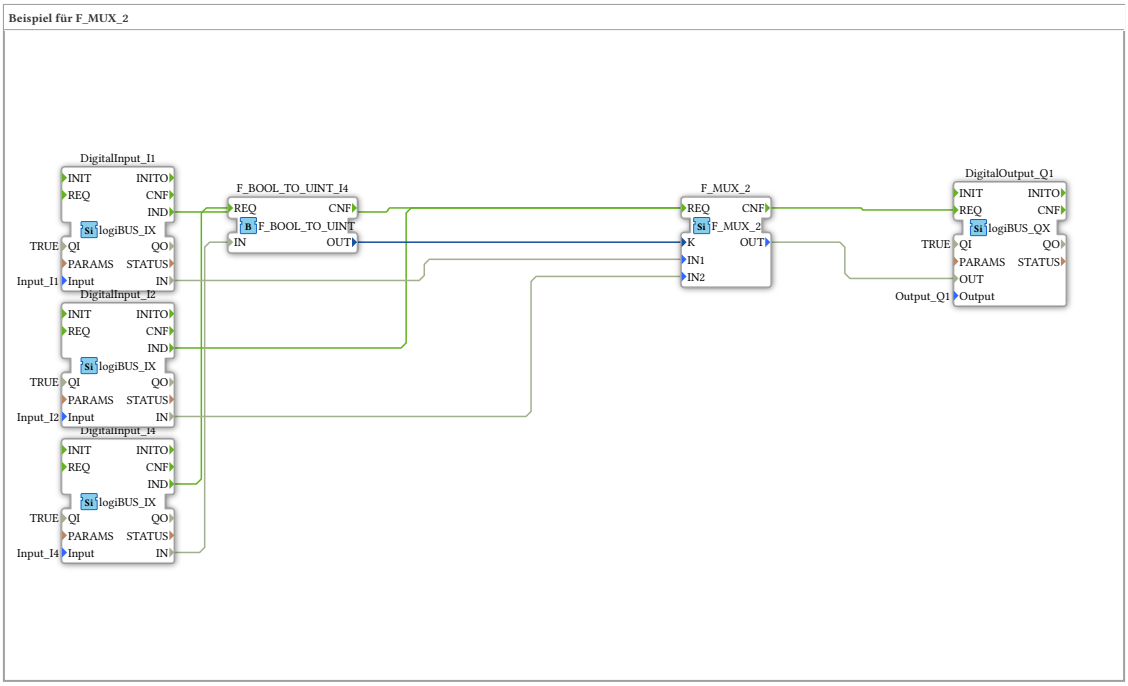
Ziel der Übung

Verwendung des Bausteins `F_MUX_2` (Multiplexer). Es wird gezeigt, wie man zwischen zwei Signalquellen umschaltet, um einen gemeinsamen Ausgang zu bedienen.

Beschreibung und Komponenten

In `Uebung_090a1.SUB` wird ein binärer Wahlschalter genutzt, um zwischen zwei Eingängen umzuschalten.

Funktionsbausteine (FBs)



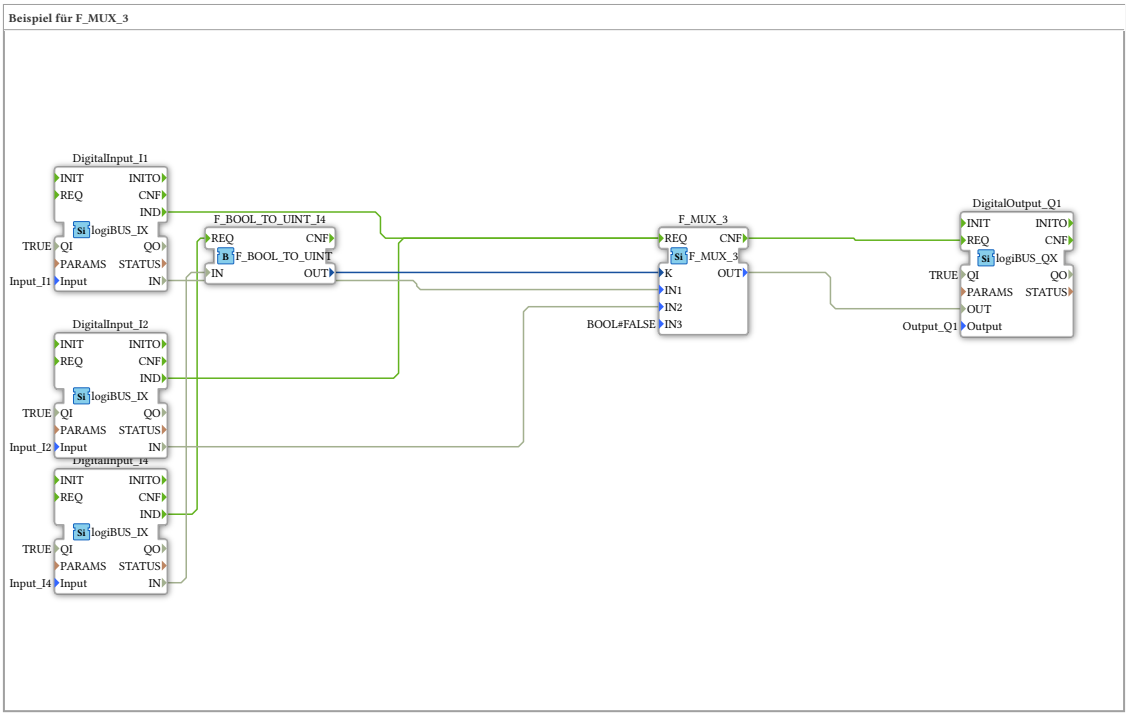
- **`I1` & `I2` (Sources)**: Die Datenquellen.
- **`I4` (Selector)**: Bestimmt, welche Quelle durchgeschaltet wird.
- **`F_MUX_2`**: Der Multiplexer-Baustein.

Funktionsweise

- Ist Taster **I4** nicht gedrückt (K=0) ➡️ Der Zustand von **I1** wird an den Ausgang `Q1` weitergereicht.
- Ist Taster **I4** gedrückt (K=1) ➡️ Der Zustand von **I2** wird an den Ausgang `Q1` weitergereicht.

Dies ermöglicht das Umschalten von Bedien-Zuständigkeiten (z.B. zwischen Lokal- und Fernsteuerung).

2.2.1.305 Uebung_090a2: Beispiel für F_MUX_3



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert die Verwendung des Multiplexer-Funktionsbausteins F_MUX_3 in Verbindung mit digitalen Ein- und Ausgängen. Die Übung zeigt, wie ein Multiplexer zur Steuerung von Ausgängen basierend auf verschiedenen Eingangssignalen eingesetzt werden kann.

Hinweis: Diese Übung enthält einen Kommentar, der darauf hinweist, dass sie derzeit nicht aufgespielt werden kann.

Verwendete Funktionsbausteine (FBs)

F_MUX_3

- **Typ**: Multiplexer mit 3 Eingängen
- **Parameter**:
  - IN3 = BOOL#FALSE (fester Wert für dritten Eingang)

DigitalInput_I1, DigitalInput_I2, DigitalInput_I4

- **Typ**: logiBUS_IX (digitale Eingänge)
- **Parameter**:
  - QI = TRUE (Qualified Input aktiviert)
  - Input = logiBUS_DI::Input_Ix (entsprechende Hardware-Eingänge)

F_BOOL_TO_UINT_I4

- **Typ**: F_BOOL_TO_UINT (Boolescher zu Unsigned Integer Konverter)

DigitalOutput_Q1

- **Typ**: logiBUS_QX (digitaler Ausgang)
- **Parameter**:
  - QI = TRUE (Qualified Output aktiviert)
  - Output = logiBUS_DO::Output_Q1 (Hardware-Ausgang Q1)

Programmablauf und Verbindungen

Ereignisverbindungen

- DigitalInput_I4.IND ⟶ F_BOOL_TO_UINT_I4.REQ
- F_MUX_3.CNF ⟶ DigitalOutput_Q1.REQ
- DigitalInput_I1.IND ⟶ F_MUX_3.REQ
- DigitalInput_I2.IND ⟶ F_MUX_3.REQ

Datenverbindungen

- F_MUX_3.OUT ⟶ DigitalOutput_Q1.OUT
- DigitalInput_I1.IN ⟶ F_MUX_3.IN1
- DigitalInput_I2.IN ⟶ F_MUX_3.IN2
- DigitalInput_I4.IN ⟶ F_BOOL_TO_UINT_I4.IN
- F_BOOL_TO_UINT_I4.OUT ⟶ F_MUX_3.K

Funktionsweise

Der Multiplexer F_MUX_3 wählt basierend auf dem Steuersignal K (von F_BOOL_TO_UINT_I4) einen der drei Eingänge aus und gibt diesen am Ausgang OUT weiter. Die digitalen Eingänge I1 und I2 werden als wählbare Eingänge verwendet, während IN3 auf FALSE festgelegt ist. Der Eingang I4 dient als Steuersignal, das über den Konverter F_BOOL_TO_UINT in den Steuereingang K des Multiplexers umgewandelt wird.

Zusammenfassung

Diese Übung veranschaulicht die grundlegende Verwendung eines Multiplexers in 4diac. Sie zeigt, wie verschiedene Eingangssignale über ein Steuersignal selektiv an einen Ausgang weitergeleitet werden können. Die Übung kombiniert digitale Ein-/Ausgänge mit Signalverarbeitungsbausteinen und Typkonvertierungen.

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

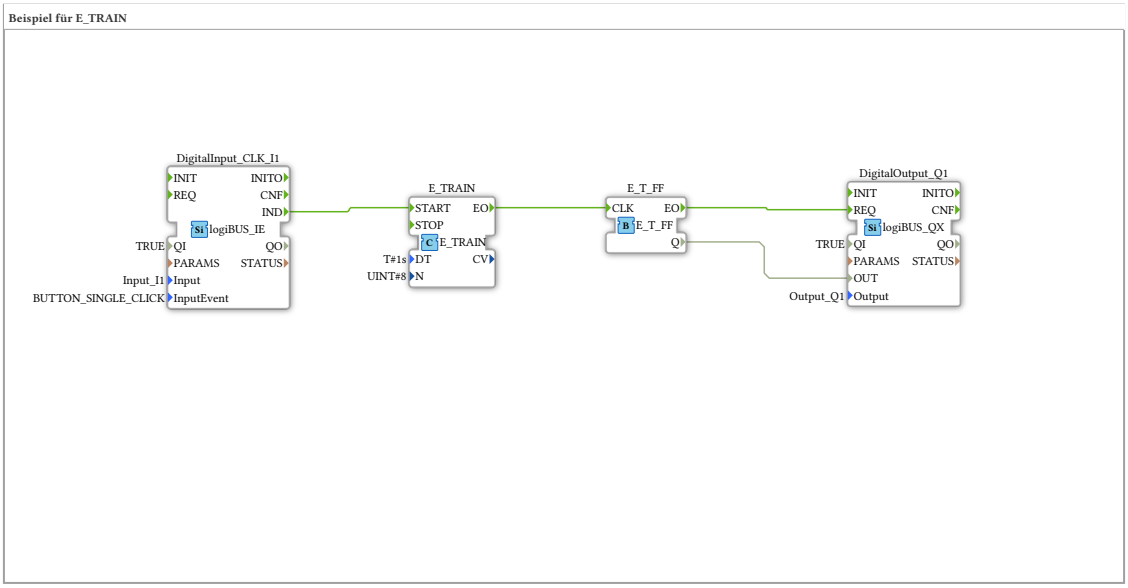
- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de

2.2.1.306 Uebung_091: Beispiel für E_TRAIN



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_091`. Hier wird die automatische Erzeugung einer festen Anzahl von Ereignissen demonstriert.

Podcast



- Als Landtechnik-Spezialist durch die Hölle: Wie Lanz-Wery Krieg, Besatzung und Hyperinflation überlebte – Einblicke in Original-Geschäftsberichte 1915-1922
- Rudolf Diesel: Geniales Werk, mysteriöses Ende – Wer verschwand 1913 auf der Fähre?
- Smart Farming Vision 1991 Auernhammers Blaupausen

Ziel der Übung

Verwendung des Bausteins `E_TRAIN`. Ziel ist es, nach einem einzelnen Start-Impuls eine definierte Folge von Ereignissen auszulösen.

Beschreibung und Komponenten

In `Uebung_091.SUB` wird ein Ereignis-Zug (Train) zur Steuerung eines Flip-Flops genutzt.

Funktionsweise

1. Der Nutzer klickt einmal auf Taster **I1**.
2. Der Baustein `E_TRAIN` startet seine Arbeit.
3. Laut Parameter `N=8` und `DT=1s` sendet der Baustein nun exakt **8 Ereignisse** im Abstand von jeweils einer Sekunde aus.
4. Diese Ereignisse gelangen an das Toggle-Flip-Flop.
5. Die Lampe an `Q1` blinkt daraufhin genau viermal (4 x An, 4 x Aus) und bleibt dann in der letzten Position stehen.

Anwendungsbeispiel

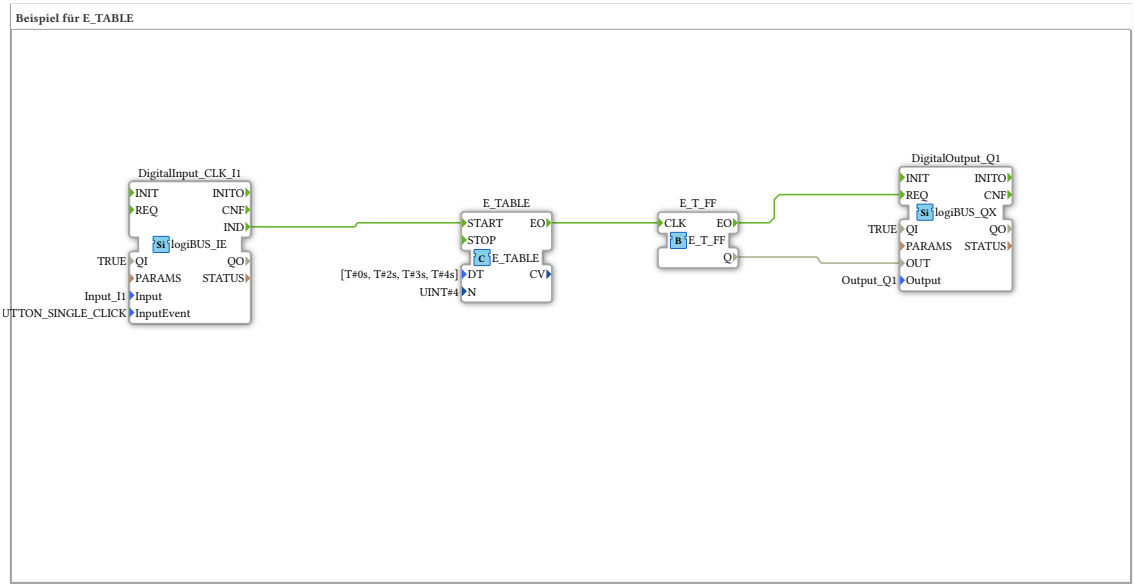
Automatisches Abkippen: Ein Hydraulikzylinder soll zum Lockern von Material dreimal kurz ruckeln. Ein Tastendruck löst die Salve von 6 Steuerbefehlen (Ausfahren-Einfahren x 3) aus, woraufhin die Steuerung den Vorgang selbstständig beendet.

2.2.1.307 Uebung_093: Beispiel für E_TABLE

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_093`. Hier wird ein komplexes Zeitmuster für Ereignisse definiert.

Podcast



- Infineon CAN-Transceiver TLE9250V versus TLE9351VSJ
- Infineon TLE9351VSJ der unsichtbare Auto-Bodyguard

Ziel der Übung

Verwendung des Bausteins `E_TABLE`. Im Gegensatz zum gleichmäßigen Takt des `E_TRAIN` erlaubt dieser Baustein die Definition von individuellen Verzögerungszeiten für jedes Ereignis in einer Liste (Array).

Beschreibung und Komponenten

In `Uebung_093.SUB` ist ein Zeit-Array hinterlegt: `[T#0s, T#2s, T#3s, T#4s]`.

Funktionsweise

Ein Klick auf **I1** startet die Tabelle:

1. Ereignis 1: Sofort (`0s`).
2. Ereignis 2: Nach weiteren 2 Sekunden.
3. Ereignis 3: Nach weiteren 3 Sekunden.
4. Ereignis 4: Nach weiteren 4 Sekunden.

Das angeschlossene Flip-Flop erzeugt somit ein unregelmäßiges Blinkmuster am Ausgang `Q1`, das exakt dem vorgegebenen Zeitplan entspricht. Dies ermöglicht die Programmierung von spezifischen Start-Sequenzen oder rhythmischen Abläufen.

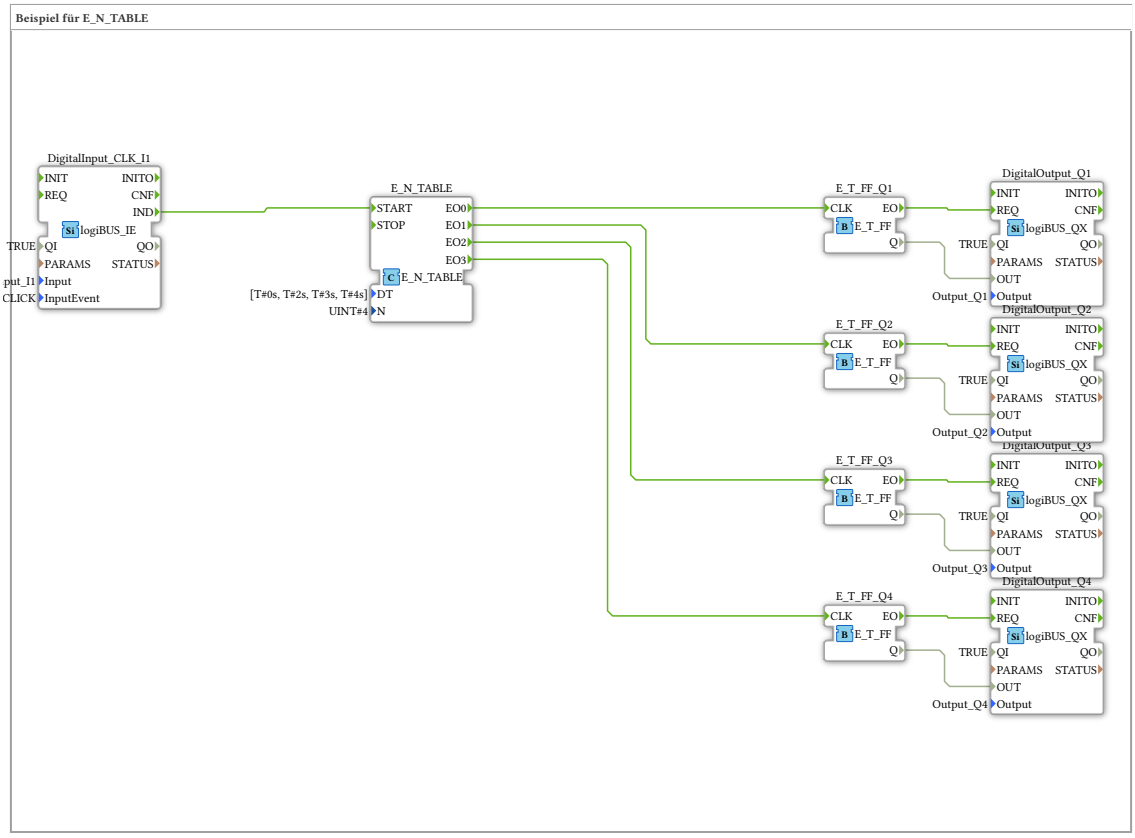
2.2.1.308 Uebung_093b: Beispiel für E_N_TABLE

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_093b`.

Übersicht

Erweiterung des Tabellen-Konzepts unter Verwendung des Bausteins `E_N_TABLE`. Anstatt alle Ereignisse an einen gemeinsamen Ausgang zu senden, verfügt dieser Baustein über separate Ausgänge (`EO0` bis `EOn`) für jeden Tabelleneintrag. In dieser Übung werden dadurch vier verschiedene Lampen (`Q1` bis `Q4`) in einer zeitlich exakt definierten, unregelmäßigen Abfolge nacheinander eingeschaltet. Dies ist die effizienteste Methode, um komplexe Start-Up-Sequenzen für Multi-Aktor-Systeme zu definieren.

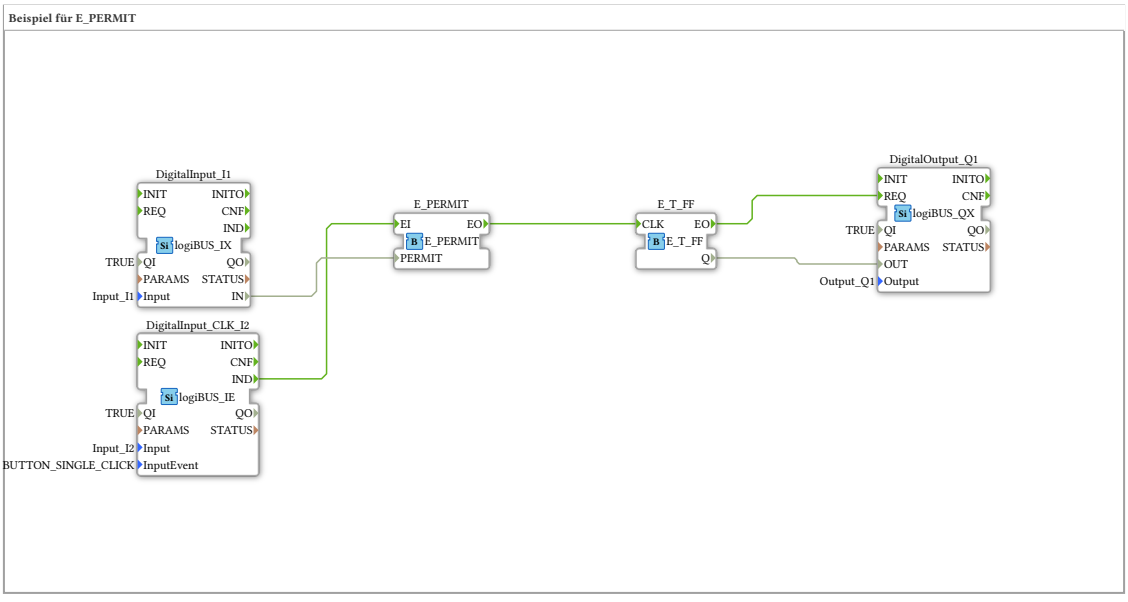


2.2.1.309 Uebung_094: Beispiel für E_PERMIT

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_094`. Hier wird eine Schutzfunktion für Ereignisströme implementiert.

🎧 Podcast



- Verfassungskunst 1946: Bayerns Bildungsauftrag zwischen Heimatliebe, Demokratie und Völkerversöhnung

---

Ziel der Übung

Verwendung des Bausteins `E_PERMIT`. Ziel ist es, die Ausführung einer Aktion (Ereignis) von einer Bedingung (Datenwert) abhängig zu machen.

Funktionsweise

Die Subapplikation `Uebung_094.SUB` nutzt einen Schalter als Freigabe für einen Taster.

- Taster **I2** liefert den Auslöse-Impuls.
- Schalter **I1** liefert die Freigabe (`PERMIT`).
- Nur wenn **I1** auf `TRUE` steht, leitet der Baustein den Klick von **I2** an das Flip-Flop weiter. Ist der Schalter aus, verpufft das Ereignis wirkungslos.

Dies ist eine einfache, aber effektive Methode zur Realisierung von Verriegelungen.

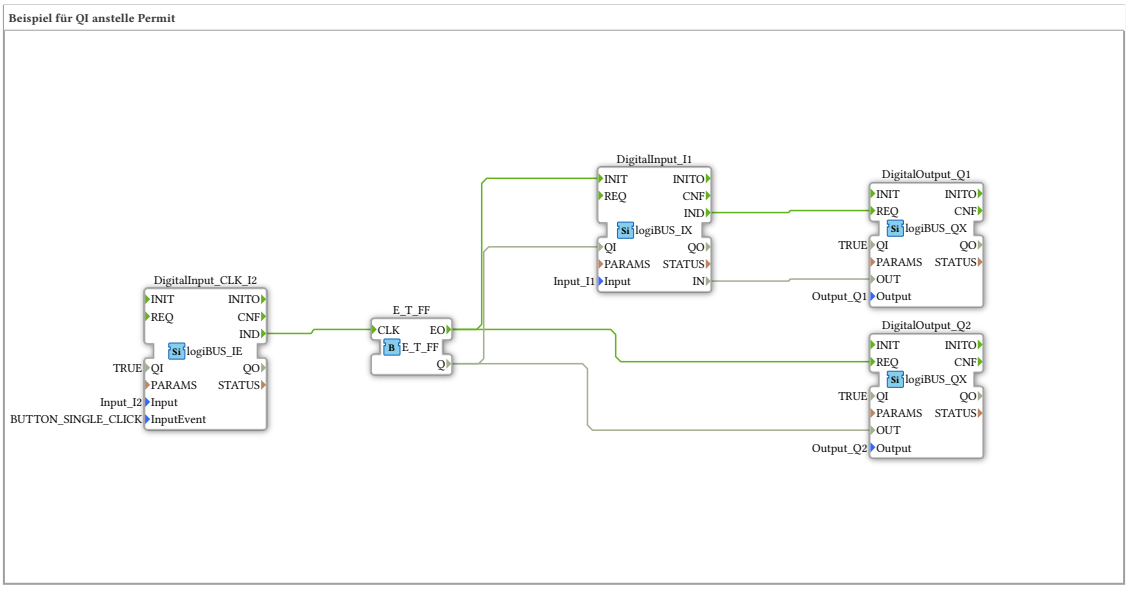
2.2.1.310 Uebung_094a: Beispiel für QI anstelle Permit



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_094a`. Hier wird eine alternative Methode zur Freigabe-Steuerung gezeigt, die direkt in den Bausteinen eingebaut ist.

Übersicht

Anstatt einen externen `E_PERMIT` Baustein zu nutzen, wird hier der Standard-Port `QI` (Qualified Input) des Eingangsbausteins `DigitalInput_I1` verwendet. Über ein Toggle-Flip-Flop wird der `QI` Eingang ein- und ausgeschaltet. Steht `QI` auf `FALSE`, ist der gesamte Baustein deaktiviert und sendet keine Ereignisse mehr an den Ausgang `Q1`, selbst wenn sich der physikalische Zustand am Hardware-Pin ändert. Dies ist die sauberste Methode, um ganze Funktionsblöcke im Programm schlafen zu legen.



2.2.1.311 Uebung_095: Beispiel für E_SELECT



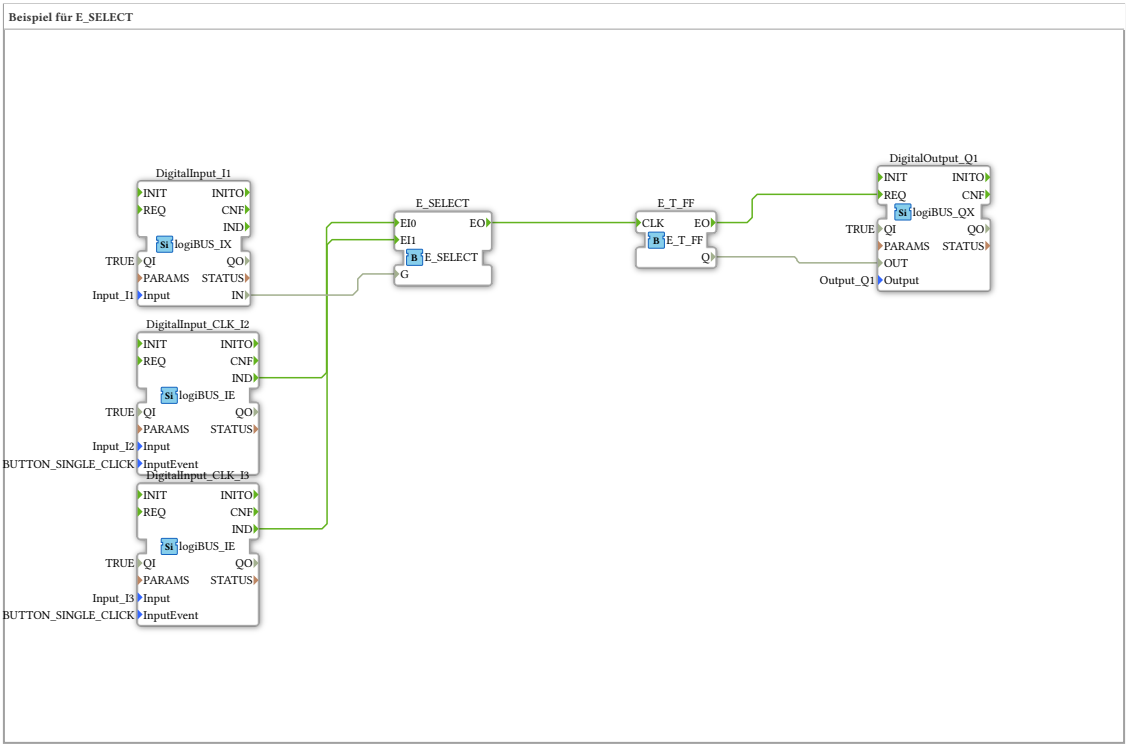
Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_095`. Hier wird die Auswahl zwischen zwei verschiedenen Ereignisquellen demonstriert.

Ziel der Übung

Verwendung des Bausteins `E_SELECT`. Dieser fungiert als Weiche für eintreffende Ereignisse („Gegenstück“ zum `E_SWITCH`).

Funktionsweise

In `Uebung_095.SUB` bestimmen zwei Taster und ein Wahlschalter die Logik.



- Schalter **I1** dient als Selektor (`G`).
- Ist **I1** auf `FALSE`, wird nur das Ereignis von Taster **I2** (`EI0`) zum Ausgang durchgelassen.
- Ist **I1** auf `TRUE`, wird nur das Ereignis von Taster **I3** (`EI1`) zum Ausgang durchgelassen.

Dies ermöglicht es, eine gemeinsame Funktion (hier das Umschalten von `Q1`) wahlweise von verschiedenen Quellen auslösen zu lassen, wobei die Steuerung aktiv festlegt, welche Quelle gerade „hörbereitschaft“ hat.

2.2.1.312 Uebung_103

Hinweis

Diese Übung existiert in den aktuellen Trainingsquellen nicht mehr (entfernt, zusammengelegt oder umbenannt). Diese Seite bleibt nur als Platzhalter für bestehende Verweise erhalten. Die aktuellen Übungen stehen in der Übungsübersicht.

2.2.1.313 Uebung_110: Übung zum Integer Überlauf

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_110`. Hier wird ein wichtiges Phänomen der digitalen Datenverarbeitung demonstriert: Der Überlauf von Variablen.

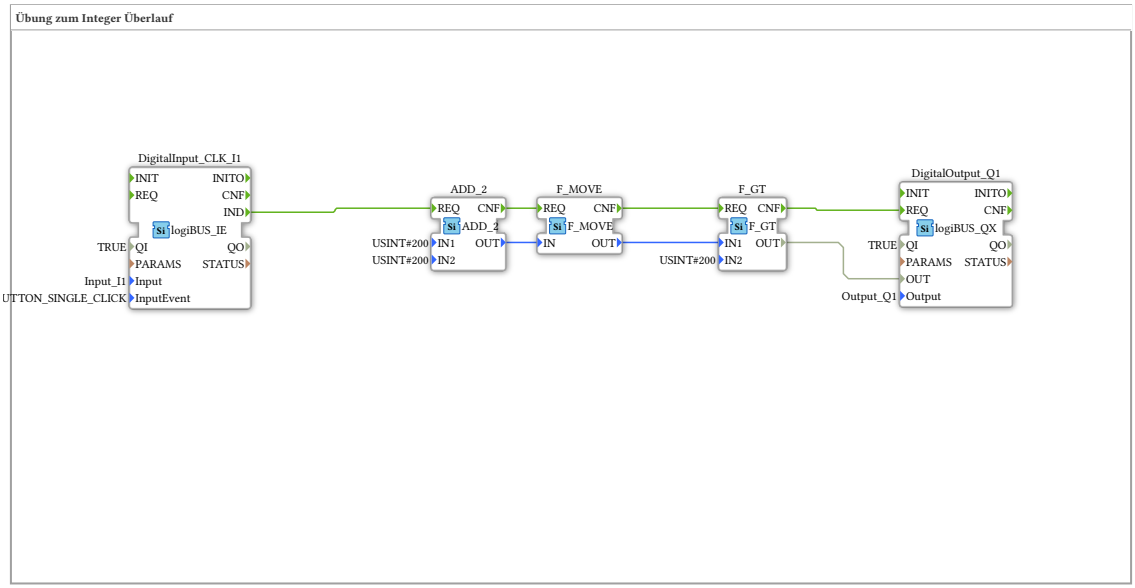
Ziel der Übung

Verständnis der Begrenzung von Datentypen. Es wird gezeigt, was passiert, wenn das Ergebnis einer Berechnung den maximalen Wertebereich eines Datentyps überschreitet.

Beschreibung und Komponenten

Die Subapplikation `Uebung_110.SUB` nutzt den Datentyp `USINT` Unsigned Short Integer. Dieser hat einen Wertebereich von 0 bis 255.

Funktionsbausteine (FBs)



- **`ADD_2`**: Addiert zwei Werte.
- **Parameter**: `IN1 = 200`, `IN2 = 200`.
- **`F_GT`**: Prüft, ob das Ergebnis größer als 200 ist.

Das Experiment

1. Mathematisch ergibt `200 + 200 = 400`.
2. Da die Variable vom Typ `USINT` aber nur bis **255** zählen kann, tritt ein Überlauf (Wrap-around) auf.
3. Das Ergebnis in der Steuerung ist `400 - 256 = 144`.
4. Der Vergleich `144 > 200` schlägt fehl (liefert `FALSE`).
5. Die Lampe an `Q1` bleibt aus, obwohl man rein rechnerisch ein „Wahr“ erwarten würde.

Fazit

Diese Übung mahnt zur Vorsicht bei der Wahl der Datentypen. Für Werte, die 255 überschreiten können, muss zwingend ein größerer Typ (z.B. `UINT` bis 65.535 oder `UDINT`) verwendet werden, um logische Fehler in der Steuerung zu vermeiden.

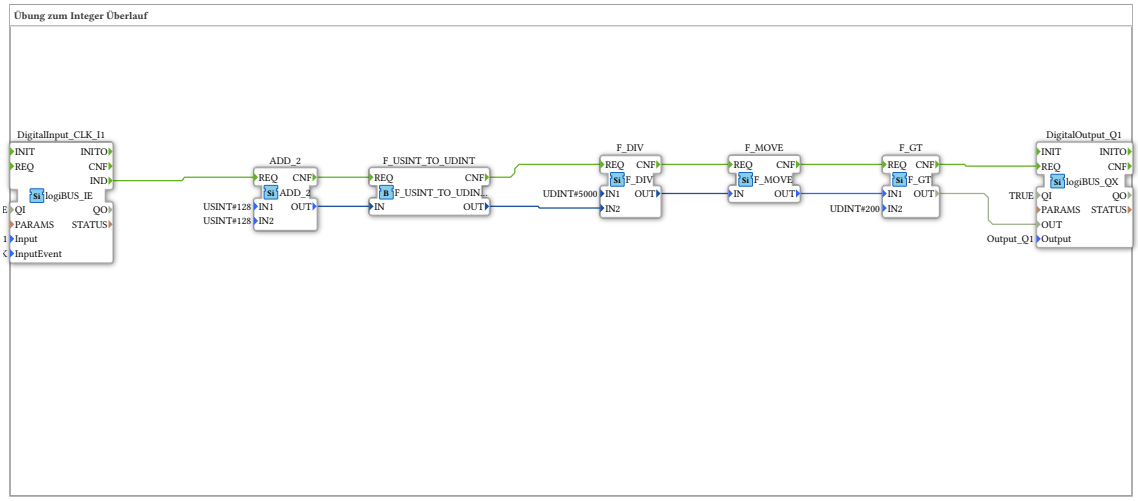
2.2.1.314 Uebung_111: Übung zum Integer Überlauf

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_111`. Hier wird gezeigt, wie man durch rechtzeitige Konvertierung in größere Datentypen Rechenfehler verhindert.

Übersicht

In `Uebung_111.SUB` wird das Überlauf-Problem aus Übung 110 gelöst. Bevor die kritische Berechnung oder der Vergleich stattfindet, wird der kleine Datentyp `USINT` über den Baustein `F_USINT_TO_UDINT` in einen großen 32-Bit-Typ gewandelt. Dadurch steht genügend „Platz“ für das Ergebnis zur Verfügung, und der anschließende Vergleich liefert das mathematisch korrekte Ergebnis. Dies demonstriert den sauberen Umgang mit verschiedenen numerischen Genauigkeiten im Programmablauf.



2.2.1.315 Uebung_120: Übung zu ISOBUS Name

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_120`. Hier wird gezeigt, wie man die Identität von Geräten im ISOBUS-Netzwerk ermittelt.

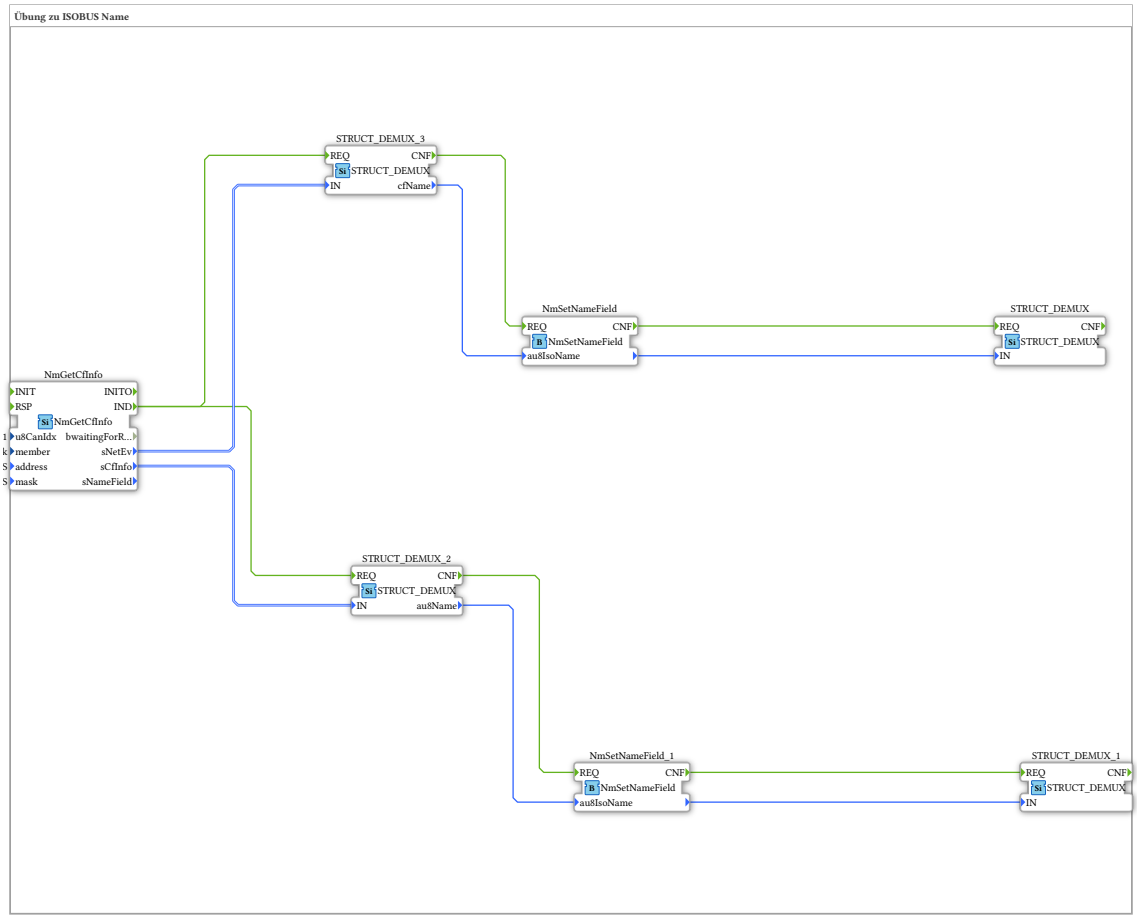
Ziel der Übung

Verwendung des Bausteins `NmGetCfInfo`. Jedes ISOBUS-Gerät besitzt einen weltweit eindeutigen 64-Bit Namen (NAME). Ziel ist es, diese Namen von anderen Teilnehmern am Bus einzulesen und in ihre Bestandteile zu zerlegen.

Beschreibung und Komponenten

In `Uebung_120.SUB` wird das Netzwerk nach aktiven Control Functions (CF) durchsucht.

Funktionsbausteine (FBs)



- **`NmGetCfInfo`**: Scant den Bus nach Teilnehmern.
- **`NmSetNameField`**: Zerlegt den 64-Bit Rohwert in die standardisierten ISOBUS-Felder.
- **`STRUCT_DEMUX`**: Macht die einzelnen Felder (Hersteller-Code, Geräteserie, Instanz etc.) für die Programmlogik zugänglich.

Funktionsweise

Der Baustein `NmGetCfInfo` liefert bei jedem Scan-Vorgang ein Datenpaket (`sCfInfo`), das den Namen eines Teilnehmers enthält. Über die nachgeschalteten Analyse-Bausteine kann das Programm nun genau „lesen“, welche Geräte (z.B. ein Terminal der Firma X oder ein Joystick der Firma Y) gerade am Traktor angeschlossen sind. Dies ist die Voraussetzung für automatisches Plug-and-Play.

2.2.1.316 Uebung_120x_sub

Hinweis

Diese Übung existiert nicht mehr als eigene Übung. Der Baustein wurde in die Bibliothek verschoben und heißt jetzt CTU_TO_PCAN_Callback. Beschreibung, Schnittstelle und Netzwerk stehen dort.

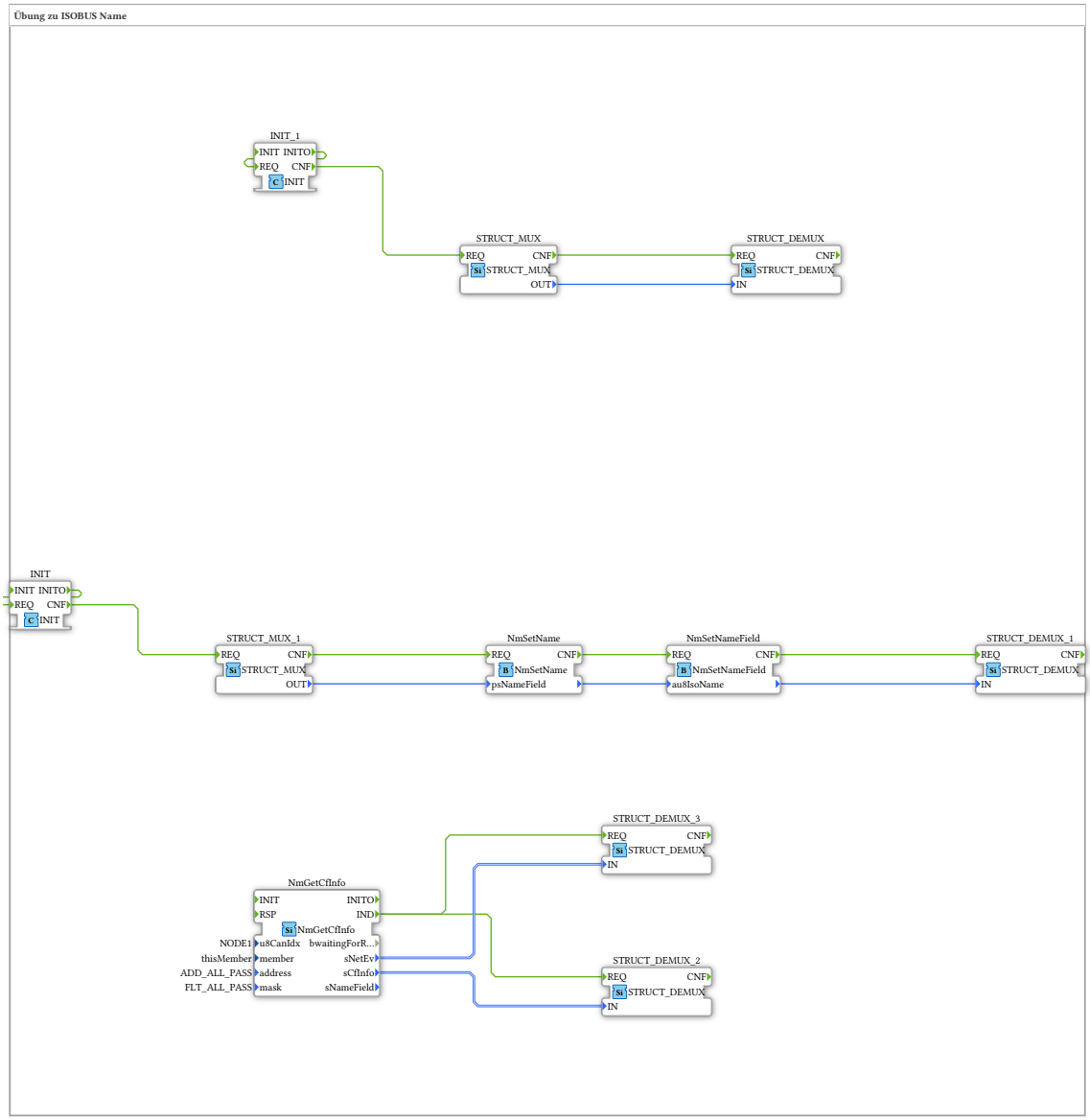
2.2.1.317 Uebung_121: Übung zu ISOBUS Name



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_121`.

Übersicht

In dieser Übung wird das Gegenstück zu 120 gezeigt: Das Auslesen der eigenen Identität. Über den Parameter `member = thisMember` am Baustein `NmGetCfInfo` liefert die Steuerung ihren eigenen 64-Bit Namen zurück. Mit dem Baustein `NmSetName` können zudem die einzelnen Felder der eigenen Identität (z.B. die Funktions-Instanz) zur Laufzeit dynamisch angepasst werden, bevor sie über den Bus kommuniziert werden.



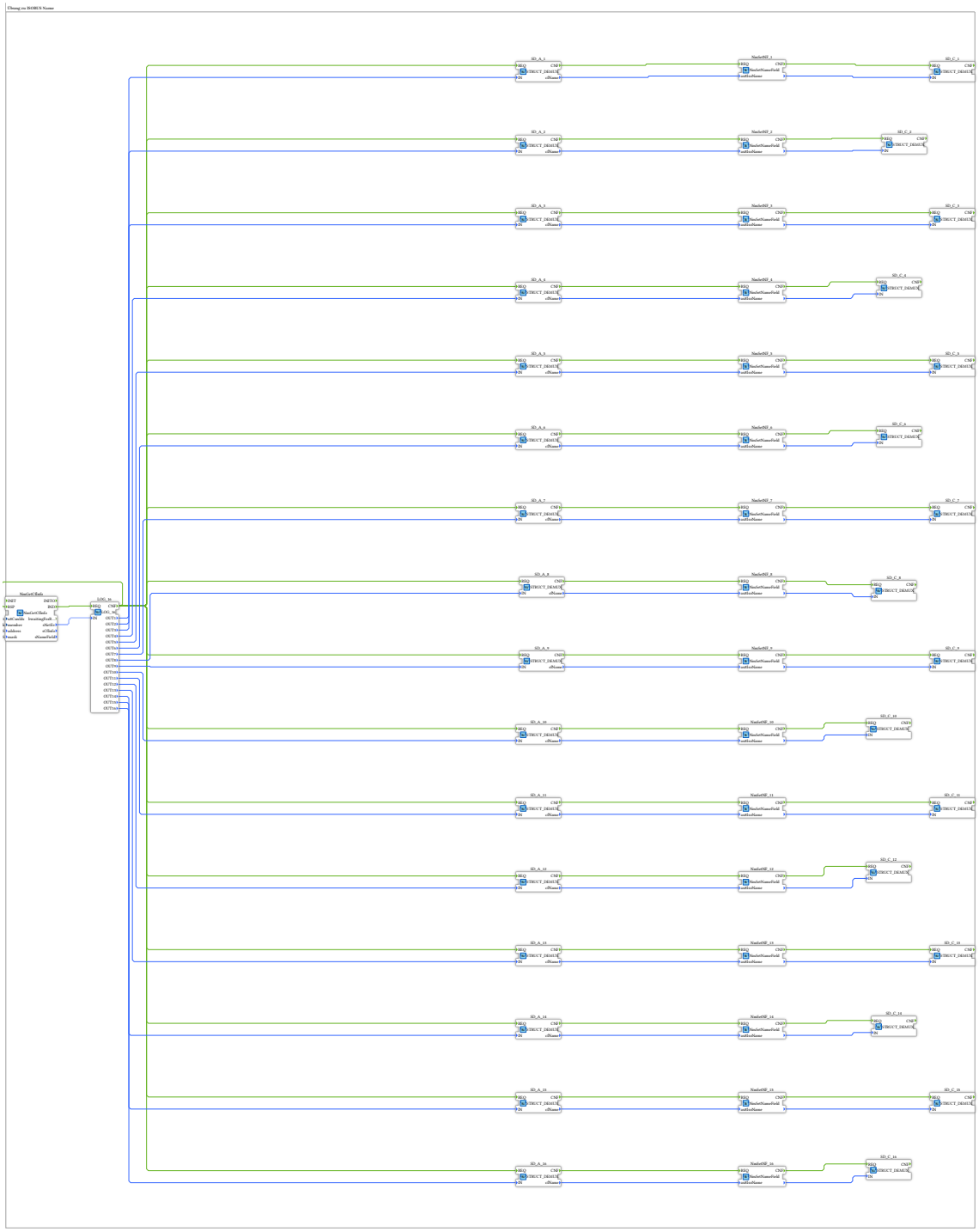
2.2.1.318 Uebung_122: Übung zu ISOBUS Name



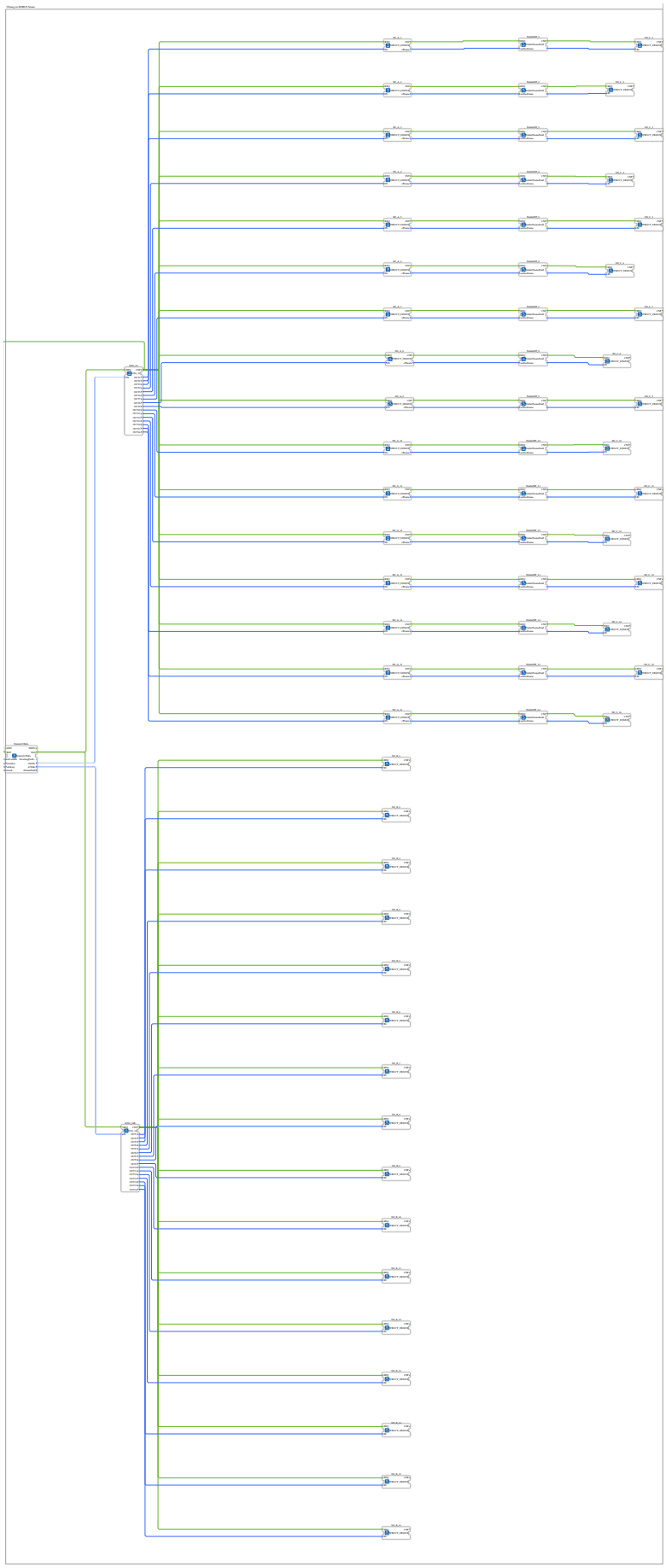
Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_122`.

Übersicht

Diese Übung demonstriert die Erfassung einer größeren Anzahl von Bus-Teilnehmern. Unter Verwendung des Bausteins `LOG_16` werden die Namen von bis zu 16 verschiedenen Control Functions im Netzwerk gleichzeitig gepuffert und analysiert. Für jeden Teilnehmer wird über eine Kette von `NmSetNameField` Bausteinen eine detaillierte Analyse der Identität durchgeführt. Dies ist ein Werkzeug für komplexe Diagnosesysteme, die den gesamten Geräteverbund eines Gespanns überwachen müssen.



2.2.1.319 Uebung_122b: Übung zu ISOBUS Name



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung („Übung zu ISOBUS Name“) beschäftigt sich mit der Analyse und Aufschlüsselung des **ISOBUS NAME**-Feldes gemäß ISO 11783. Ziel ist es, Informationen über Teilnehmer (Control Functions - CF) am Bus abzurufen, deren 64-Bit-Namen zu extrahieren und diesen Namen in seine einzelnen Bestandteile (wie Hersteller, Geräteklasse, Funktion usw.) zu zerlegen.

Die Übung ist als Sub-Application (`SubAppType`) realisiert und verarbeitet Listen von Netzwertereignissen und CF-Informationen.

Verwendete Funktionsbausteine (FBs)

In dieser Sub-Application werden verschiedene Funktionsbausteine instanziiert, um die Datenverarbeitung und -visualisierung zu realisieren.

Haupt-Bausteine

1. NmGetCfInfo (`isobus::pgn::NmGetCfInfo`)

Dieser Baustein ist der Einstiegspunkt der Übung. Er ruft Informationen über die Control Functions (CF) im Netzwerk ab.

- **Parameter**:
  - `u8CanIdx` = `NODE1` (CAN-Knoten 1)
  - `member` = `intern`
  - `address` = `FLT_ALL_PASS` (Filter: Alle Adressen)
  - `mask` = `FLT_ALL_PASS`
- **Funktionsweise**: Er liefert Arrays von Netzwerkereignissen (`sNetEv`) und CF-Informationen (`sCfInfo`), die anschließend verarbeitet werden.

2. LOG_16 (`logiBUS::utils::logging::LOG_16`)

Hier werden zwei Instanzen (`LOG_16` und `LOG_16B`) verwendet.

- **Funktionsweise**: Diese Bausteine dienen in dieser Übung als „Splitter“ oder De-Multiplexer für Arrays. Sie nehmen die Listen (Arrays mit bis zu 16 Einträgen) von `NmGetCfInfo` entgegen und geben die einzelnen Elemente an 16 separaten Ausgängen aus. Dies ermöglicht die parallele Verarbeitung der ersten 16 erkannten Geräte.

3. STRUCT_DEMUX (`eclipse4diac::convert::STRUCT_DEMUX`)

Dieser generische Konvertierungsbaustein wird vielfach eingesetzt (`SD_A_x`, `SD_B_x`, `SD_C_x`), um komplexe Datentypen (Strukturen) in ihre Einzelteile zu zerlegen, damit diese visualisiert oder weiterverarbeitet werden können.

- **Verwendete Typen**:
  - `isobus::pgn::ISONETEVENT_T` (bei `SD_A_x`): Extrahiert u.a. den rohen `cfName`.
  - `isobus::pgn::CF_INFO_T` (bei `SD_B_x`): Zeigt Statusinformationen der CF an.
  - `isobus::pgn::NAMEFIELD_T` (bei `SD_C_x`): Zeigt die dekodierten Felder des ISOBUS Namens an.

4. NmSetNameField (`isobus::pgn::NmSetNameField`)

Dies ist der Kernbaustein für die Interpretation des Namens. Er kommt 16-mal vor (`NmSetNF_1` bis `NmSetNF_16`).

- **Eingang**: `au8IsoName` (Der 64-Bit ISOBUS Name als Byte-Array).
- **Funktionsweise**: Der Baustein analysiert den ISOBUS-Namen und schlüsselt ihn gemäß der Norm in eine Struktur (`NAMEFIELD_T`) auf. Diese enthält Informationen wie:
  - Identity Number
  - Manufacturer Code
  - ECU Instance
  - Function Instance
  - Function
  - Vehicle System
  - Industry Group
  - Arbitrary Address Capable

Programmablauf und Verbindungen

Der Ablauf der Übung lässt sich in drei parallele Verarbeitungsstränge unterteilen, die durch das Triggern von `NmGetCfInfo` angestoßen werden:

1. **Erfassung (NmGetCfInfo)**: Der Baustein scannt den Bus und gibt bei Ereignissen (`IND`) die aktuellen Listen der Netzwerkteilnehmer aus.
2. **Verteilung (LOG_16 & LOG_16B)**: Die Ausgänge `sNetEv` (Network Events) und `sCfInfo` (Control Function Infos) werden an die `LOG_16`-Bausteine übergeben. Diese brechen die Arrays auf einzelne Verbindungen herunter (Index 1 bis 16).
3. **Verarbeitungspfad A & C (Namens-Analyse)**:
   - Die einzelnen Netzwerkereignisse gehen vom `LOG_16` zu den `SD_A`-Bausteinen.
   - Dort wird das Attribut `cfName` (der ISOBUS Name) extrahiert.
   - Dieser `cfName` wird direkt an den jeweiligen `NmSetNF`-Baustein weitergeleitet.
   - Der `NmSetNF`-Baustein dekodiert den Namen.
   - Das Ergebnis (die Struktur mit den lesbaren Feldern) wird im `SD_C`-Baustein aufgeschlüsselt angezeigt. So kann man z.B. sehen, welcher Hersteller hinter einem Gerät steckt.
4. **Verarbeitungspfad B (Info-Anzeige)**:
   - Parallel dazu werden die allgemeinen CF-Informationen vom `LOG_16B` an die `SD_B`-Bausteine geleitet. Dies dient vermutlich der Diagnose von Adressen und Status der Teilnehmer, unabhängig von der Namens-Dekodierung.

Zusammenfassung

Die Übung **Uebung_122b** demonstriert die Detailanalyse von ISOBUS-Teilnehmern. Durch die Kombination von Listenabruf, De-Multiplexing und spezifischen Parsing-Bausteinen (`NmSetNameField`) wird gezeigt, wie aus dem kryptischen 64-Bit-Namen eines Steuergerätes (ECU) menschenlesbare Informationen wie Hersteller, Geräteklasse und Funktion extrahiert werden können. Dies ist essenziell für Diagnoseanwendungen und die Interoperabilität im ISOBUS-Netzwerk.

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

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

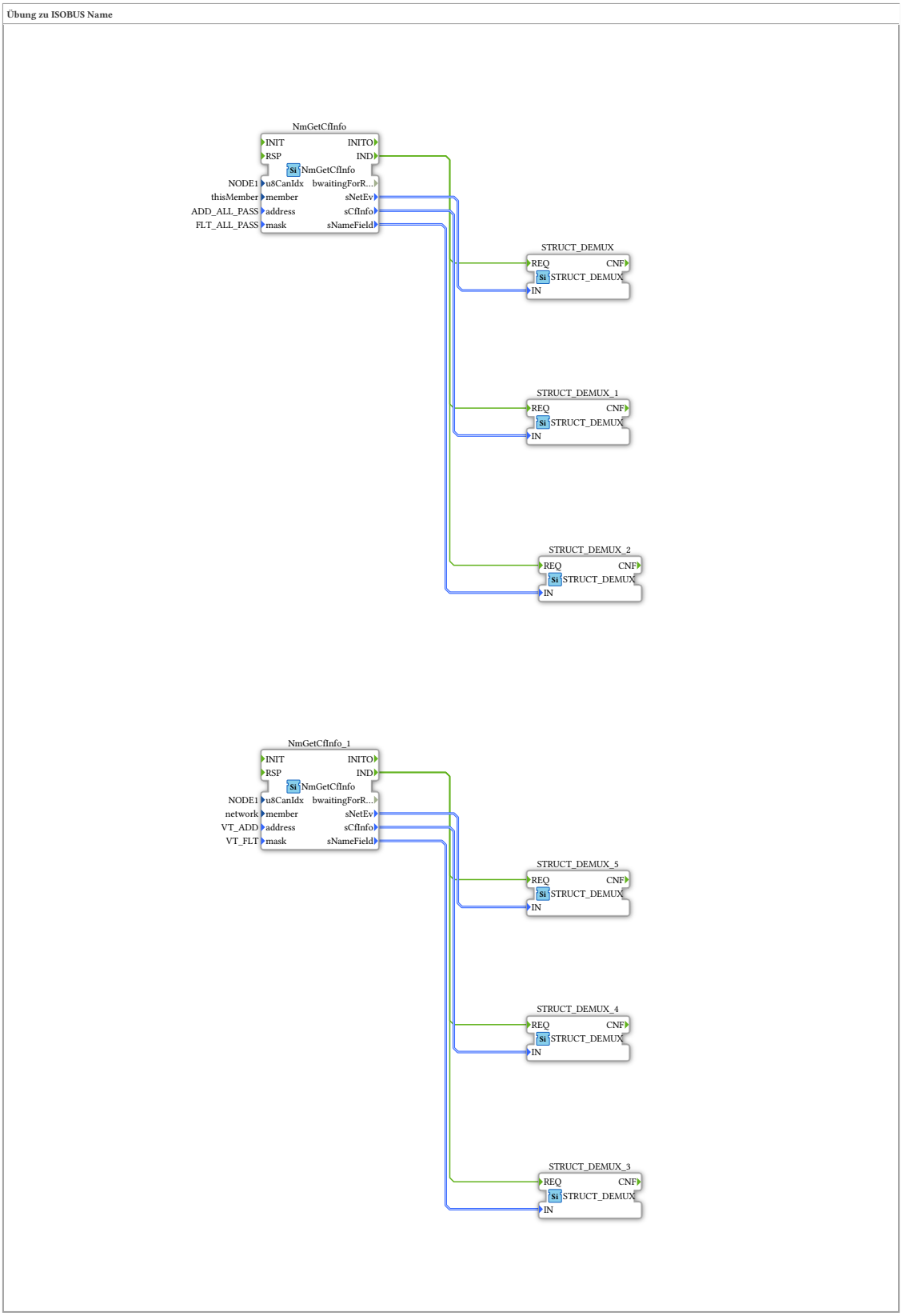
2.2.1.320 Uebung_123: Übung zu ISOBUS Name



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_123`.

Übersicht

Hier wird gezeigt, wie man den Netzwerk-Scan gezielt auf bestimmte Gerätetypen einschränkt. Durch die Angabe einer Zieladresse (`VT_ADD`) und einer Maske (`VT_FLT`) am Baustein `NmGetCfInfo_1` wird das Programm so konfiguriert, dass es nur auf Teilnehmer reagiert, die dem Profil eines „Virtual Terminals“ (VT) entsprechen. Alle anderen Bus-Teilnehmer (z.B. Joysticks oder Task Controller) werden von diesem Baustein ignoriert. Dies ermöglicht eine saubere Trennung der Kommunikationslogik nach Funktionsgruppen.


Übung zu ISOBUS Name
NmGetCfInfo
INIT
RSP
INITO
IND
NmGetCfInfo
NODE1
u8CanIdx
bwaitingForR...
thisMember
member
sNetEv
ADD_ALL_PASS
address
sCfInfo
FLT_ALL_PASS
mask
sNameField
STRUCT_DEMUX
REQ
CNF
STRUCT_DEMUX
IN
STRUCT_DEMUX_1
REQ
CNF
STRUCT_DEMUX
IN
STRUCT_DEMUX_2
REQ
CNF
STRUCT_DEMUX
IN
NmGetCfInfo_1
INIT
RSP
INITO
IND
NmGetCfInfo
NODE1
u8CanIdx
bwaitingForR...
network
member
sNetEv
VT_ADD
address
sCfInfo
VT_FLT
mask
sNameField
STRUCT_DEMUX_5
REQ
CNF
STRUCT_DEMUX
IN
STRUCT_DEMUX_4
REQ
CNF
STRUCT_DEMUX
IN
STRUCT_DEMUX_3
REQ
CNF
STRUCT_DEMUX
IN

2.2.1.321 Uebung_124: Übung zu ISOBUS Send Message

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_124`. Hier verlassen wir die Standard-Nachrichten und senden eigene Datenpakete (PGNs) an einen spezifischen Partner im Netzwerk.

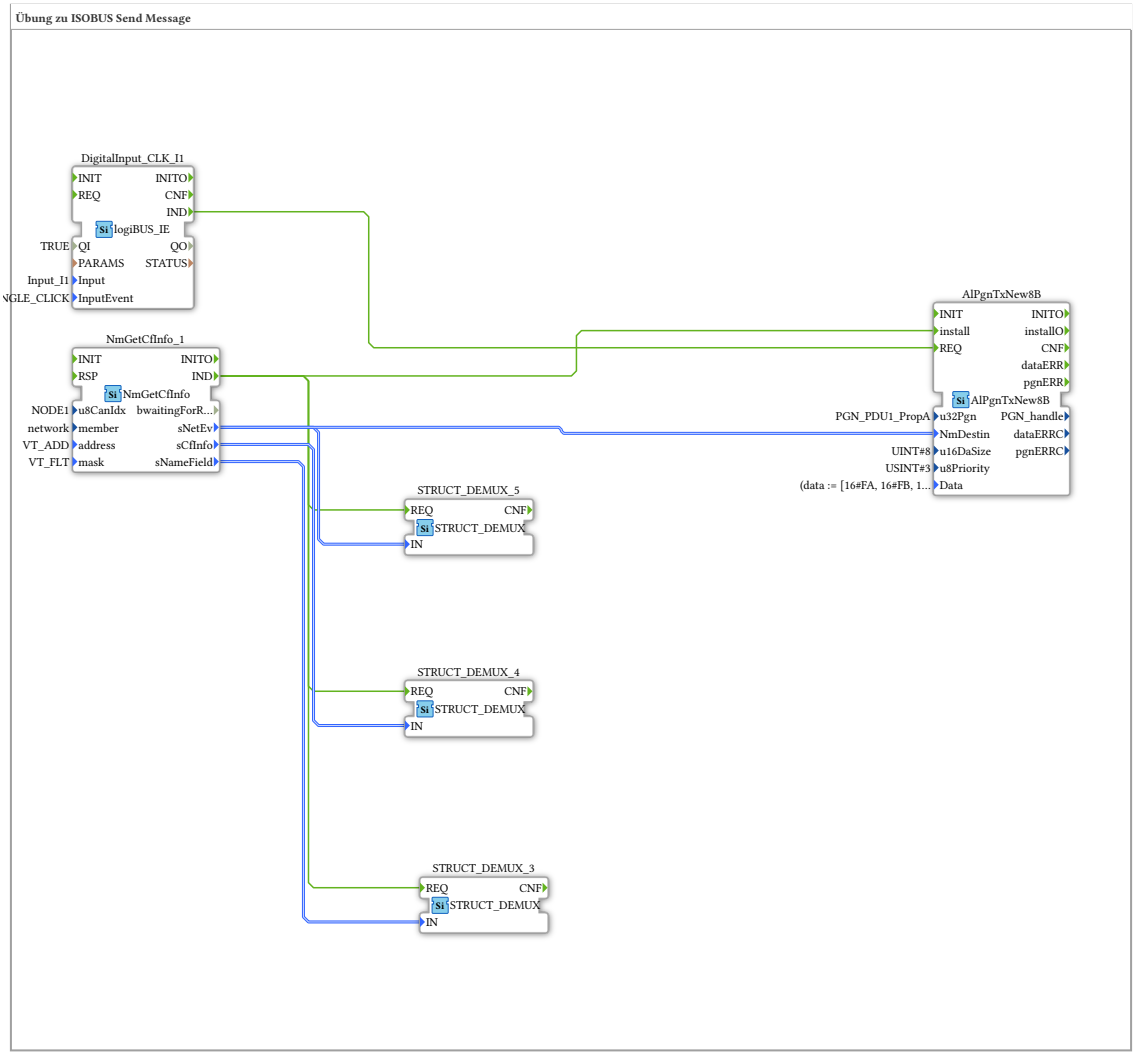
Ziel der Übung

Verwendung des Bausteins `AlPgnTxNew8B`. Es wird gezeigt, wie man eine herstellerspezifische Nachricht (Proprietary PGN) definiert und diese gezielt an ein anderes Steuergerät sendet.

Beschreibung und Komponenten

Die Subapplikation `Uebung_124.SUB` kombiniert die Teilnehmer-Suche mit einem Sende-Baustein.

Funktionsbausteine (FBs)



- **`NmGetCfInfo_1`**: Sucht den Zielpartner (hier ein Virtual Terminal).
- **`AlPgnTxNew8B`**: Der Sende-Baustein für 8-Byte Nachrichten.
- **Parameter**:
  - `u32Pgn`: Die Nummer der Nachricht (hier `61184` = Proprietary A).
  - `Data`: Der Inhalt der Nachricht (8 Bytes Hex-Daten).

Funktionsweise

1. Zuerst identifiziert `NmGetCfInfo` den Partner und liefert dessen Netzwerk-Identität (`NmDestin`).
2. Durch das `IND`-Event wird der Sende-Baustein einmalig im System registriert (`install`).
3. Jeder Klick auf den physischen Taster **I1** triggert nun den `REQ`-Eingang.
4. Die Steuerung sendet daraufhin das vordefinierte Datenpaket direkt an den gewählten Partner.

Dies ist die Grundlage für die herstellerspezifische Kommunikation zwischen Traktor und Anbaugerät.

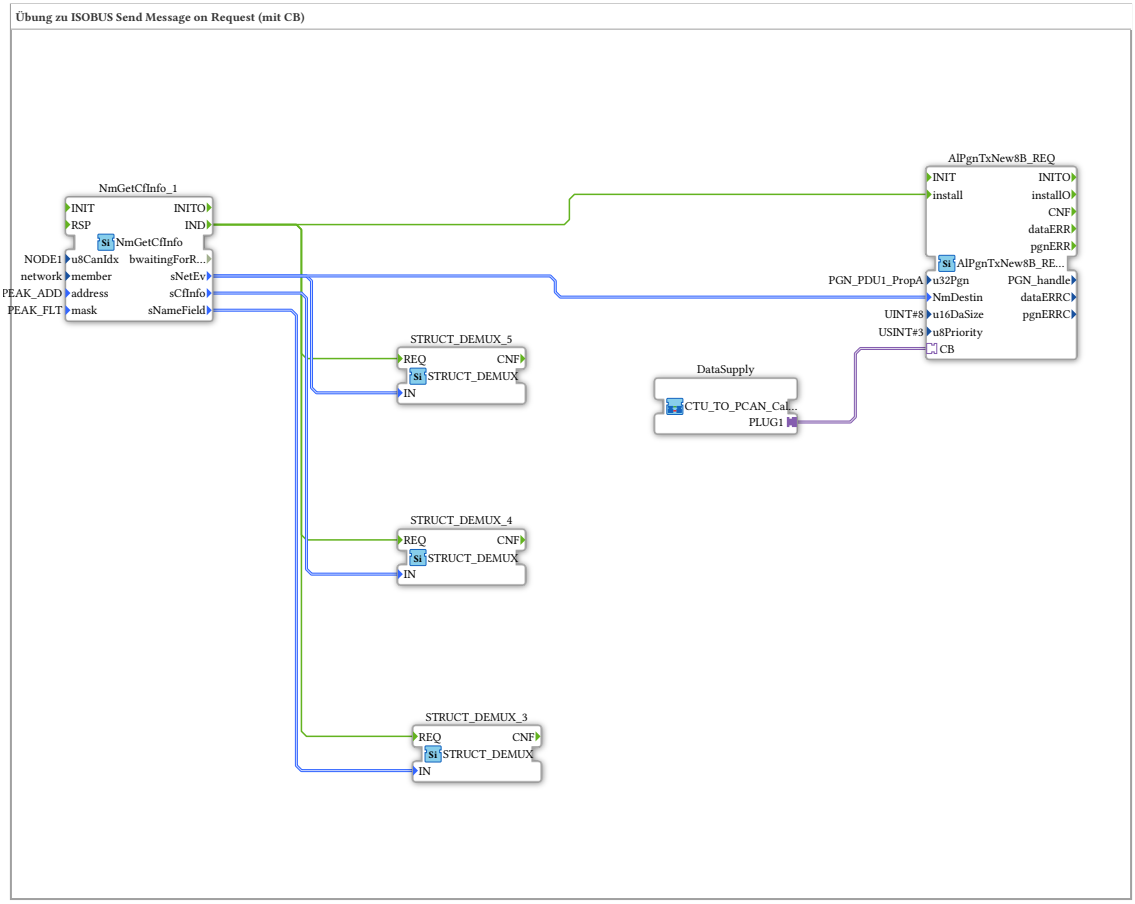
2.2.1.322 Uebung_125: Übung zu ISOBUS Send Message on Request (mit CB)



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_125`.

Übersicht

Verwendung des Bausteins `AlPgnTxNew8B_REQ` mit Callback-Funktion. In dieser Übung sendet die Steuerung die Daten nicht von sich aus, sondern wartet passiv auf eine Anfrage (ISO Request) eines anderen Teilnehmers. Sobald eine Anfrage für die spezifizierte PGN (`61184`) eintrifft, holt sich der Baustein die aktuellen Daten über den Adapter-Port `CB` (Callback) aus der Sub-Applikation `DataSupply` und sendet die Antwort automatisch zurück. Dies spart Buslast, da Daten nur bei echtem Bedarf übertragen werden.



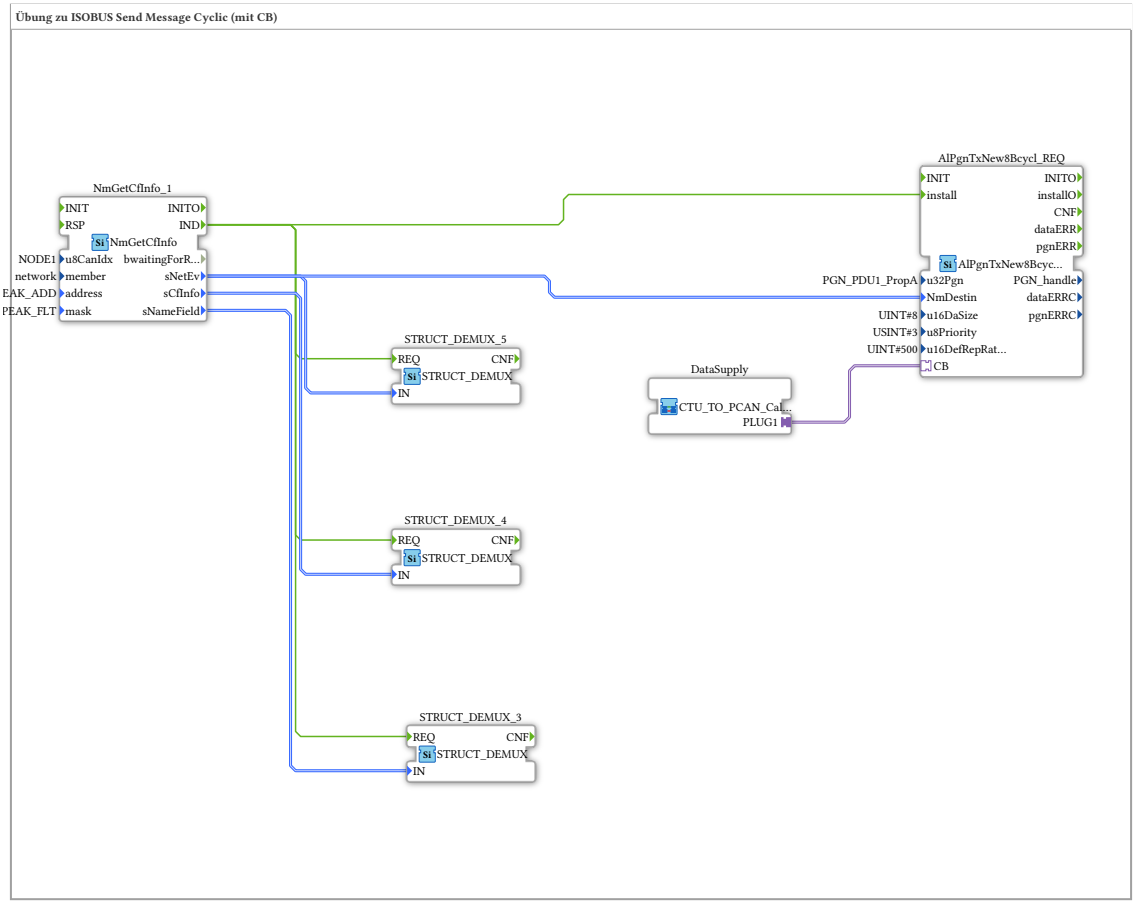
2.2.1.323 Uebung_126: Übung zu ISOBUS Send Message Cyclic (mit CB)



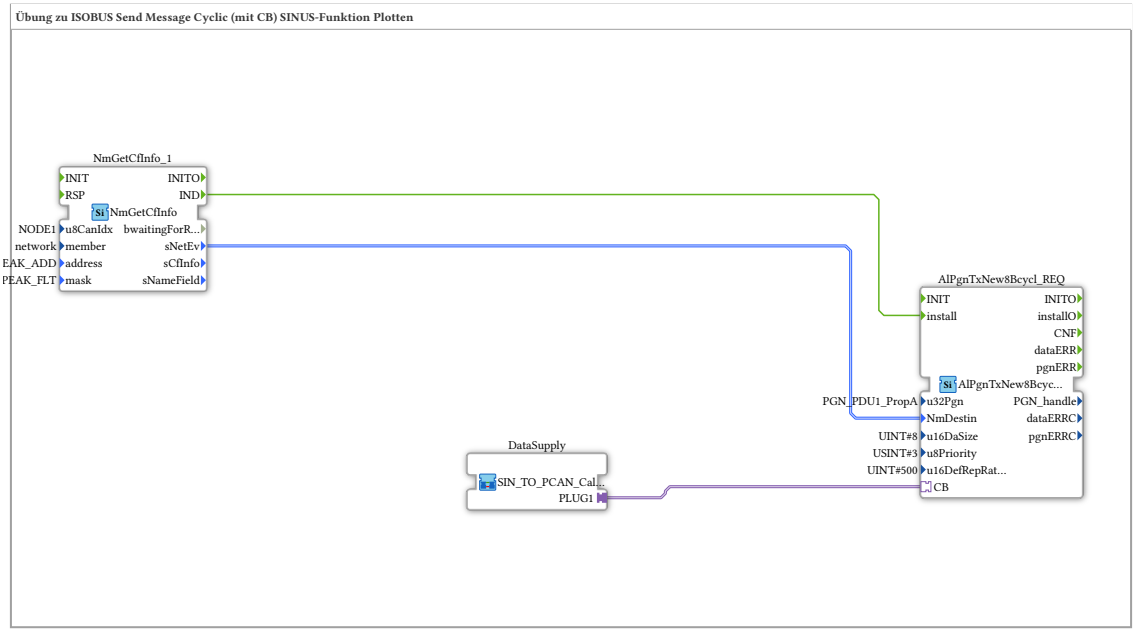
Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_126`.

Übersicht

Verwendung des Bausteins `AlPgnTxNew8Bcycl_REQ`. Hier wird die Nachricht in einem festen Zeitintervall (Parameter `u16DefRepRate = 500`ms) wiederholt auf den Bus gesendet. Der Baustein nutzt ebenfalls den Callback-Mechanismus (`CB`), um vor jedem Sendevorgang die allerneuesten Daten aus der Applikation abzufragen. Dies ist das Standard-Verfahren für Status-Nachrichten, die permanent zur Verfügung stehen müssen (z.B. Herzschlag oder Sensordaten).



2.2.1.324 Uebung_126b: Übung zu ISOBUS Send Message Cyclic (mit CB) SINUS-Funktion Plotten



• – ◦ • – ◦ • – ◦ •

Einleitung

Die Übung **Uebung_126b** demonstriert das zyklische Senden einer ISOBUS-Nachricht, bei der die Nutzdaten (Data Payload) dynamisch über einen Callback-Mechanismus generiert werden. Konkret wird eine Sinus-Funktion erzeugt, deren Werte in das erste Byte der CAN-Nachricht geschrieben werden. Dies dient beispielsweise dazu, Signalverläufe zu simulieren und diese anschließend in Diagnose-Tools wie dem PCAN-Explorer zu plotten.

Das Hauptmerkmal dieser Übung ist die Trennung von Kommunikationsmanagement (im Haupt-Netzwerk) und Datengenerierung (in einer Sub-Application), verbunden über einen Adapter.

Verwendete Funktionsbausteine (FBs)

Haupt-Netzwerk

Im Hauptnetzwerk werden folgende Bausteine verwendet, um die Kommunikation zu initiieren:

- **NmGetCfInfo_1** (`isobus::pgn::NmGetCfInfo`):
  - Dient zum Abrufen von Netzwerkinformationen und Filtern der Zieladresse.
  - Parameter:
    - `u8CanIdx` = `NODE1`
    - `member` = `network`
    - `address` = `PEAK_ADD`
    - `mask` = `PEAK_FLT`
- **AlPgnTxNew8Bcycl_REQ** (`isobus::pgn::tx::AlPgnTxNew8Bcycl_REQ`):
  - Dieser Baustein ist für das zyklische Senden einer 8-Byte großen Nachricht zuständig.
  - Er verfügt über einen Adapter-Anschluss (`CB`), über den die Datenanforderung (Callback) abgewickelt wird.
  - Parameter:
    - `u32Pgn` = `61184` (Die verwendete Parameter Group Number)
    - `u16DaSize` = `8` (Datenlänge in Bytes)
    - `u8Priority` = `3` (Priorität der Nachricht)
    - `u16DefRepRate` = `500` (Zykluszeit in Millisekunden)
- **DataSupply** (`Uebungen::Uebung_126b_sub`):
  - Eine Sub-Application, welche die Logik zur Datenerzeugung (Sinus-Kurve) enthält.

Sub-Bausteine: DataSupply (Uebung_126b_sub)

Diese Sub-Application kapselt die Logik zur Berechnung der Sinuswerte.

- **Typ**: SubApp
- **Verwendete interne FBs**:
  - **CallbackFB**: `isobus::pgn::tx::CallbackFB`
    - Dieser Baustein dient als Schnittstelle zum zyklischen Sender. Wenn der Sender Daten benötigt, triggert dieser FB die Berechnungskette.
    - Parameter: `DI1` (Eingang für die berechneten Datenstruktur)
  - **GEN_SIN**: `OSCAT::Basic::POUs::Engineering::signal_generators::GEN_SIN`
    - Erzeugt ein sinusförmiges Signal.
    - Parameter:
      - `PT` = `T#10s` (Periodendauer)
      - `AM` = `10.0` (Amplitude)
      - `OS` = `5.0` (Offset/Verschiebung)
      - `DL` = `0.0`
  - **F_LREAL_TO_USINT**: `iec61131::conversion::F_LREAL_TO_USINT`
    - Konvertiert den Gleitkommawert des Sinus-Generators in eine vorzeichenlose kleine Ganzzahl (USINT).
  - **F_USINT_TO_BYTE**: `iec61131::conversion::F_USINT_TO_BYTE`
    - Wandelt den USINT-Wert in ein Byte um.
  - **BYTES_TO_ARR08B**: `logiBUS::utils::conversion::arr::reversing::BYTES_TO_ARR08B`
    - Erstellt ein Byte-Array aus 8 Einzelbytes.
    - Der berechnete Sinus-Wert wird an `IN_00` angelegt.
    - Parameter `IN_01` bis `IN_07` sind statisch `16#00`.
  - **STRUCT_MUX**: `eclipse4diac::convert::STRUCT_MUX`
    - Verpackt das Byte-Array in die `isobus::pgn::CAN_MSG` Struktur.
    - Attribut: `StructuredType` = `isobus::pgn::CAN_MSG`
- **Funktionsweise**: Sobald der `CallbackFB` ein Event empfängt (getriggert durch den zyklischen Sender im Hauptprogramm), aktiviert er den `GEN_SIN`. Der aktuelle Sinuswert wird berechnet, konvertiert (LREAL -> USINT -> BYTE) und in das erste Byte eines Arrays geschrieben. Dieses Array wird in eine CAN-Nachricht-Struktur verpackt und über den `CallbackFB` zurück an das Hauptprogramm gegeben.

Programmablauf und Verbindungen

1. **Initialisierung**: Der Baustein `NmGetCfInfo_1` ermittelt beim Start die notwendigen Netzwerkinformationen.
2. **Konfiguration**: Sobald die Netzwerkinformationen verfügbar sind (`IND` Event), wird der Senderbaustein `AlPgnTxNew8Bcycl_REQ` installiert (`install`).
3. **Zyklischer Betrieb**:
   - Der `AlPgnTxNew8Bcycl_REQ` Baustein ist auf eine Wiederholrate von 500ms eingestellt.
   - Alle 500ms löst er über den Adapter-Anschluss `CB` (verbunden mit `DataSupply.PLUG1`) eine Anfrage aus.
4. **Datenberechnung**:
   - Innerhalb der Sub-App `DataSupply` empfängt der `CallbackFB` die Anfrage.
   - Dies triggert die Signalkette: Der Sinus-Generator `GEN_SIN` berechnet den nächsten Wert basierend auf der aktuellen Zeit.
   - Aufgrund der Parameter (Amplitude 10, Offset 5) erzeugt der Generator Werte im Bereich von −5.0 bis +15.0. Da die Konvertierung auf `USINT` erfolgt, werden negative Werte typischerweise auf 0 geklemmt.
5. **Rückgabe und Senden**:
   - Der berechnete Wert landet im ersten Byte der Nutzdaten.
   - Die Daten werden über den Adapter zurück an `AlPgnTxNew8Bcycl_REQ` geleitet.
   - Der Baustein sendet die PGN 61184 mit den aktuellen Daten auf den CAN-Bus.

Lernziele:

- Verständnis des Adapter-Konzepts (Plugs/Sockets) in 4diac.
- Nutzung von Callback-Mechanismen für „Just-in-Time“ Datenerzeugung bei zyklischen Sendern.
- Verwendung von OSCAT-Bibliotheksbausteinen (`GEN_SIN`) zur Signalsimulation.

- Datenkonvertierung und Strukturierung für ISOBUS/CAN-Nachrichten.

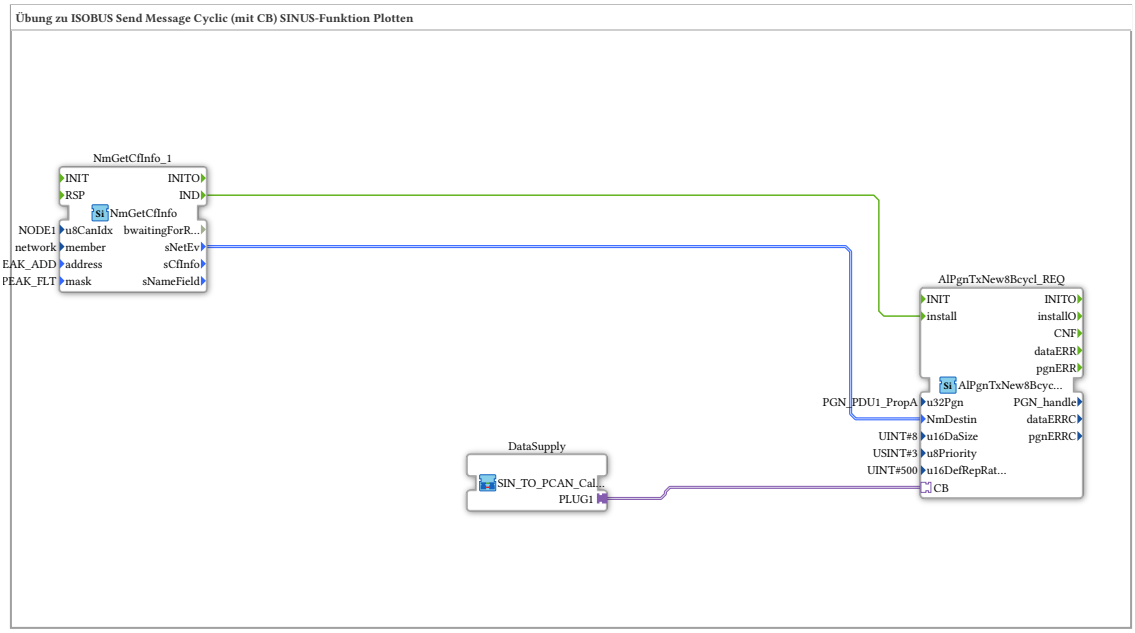
Zusammenfassung

Die Übung 126b zeigt eine elegante Methode, um Simulationsdaten (hier eine Sinuskurve) über den ISOBUS zu senden. Durch die Auslagerung der Datenerzeugung in eine Sub-Applikation und die Nutzung der Callback-Schnittstelle bleibt die Hauptanwendung übersichtlich und der zyklische Sender kümmert sich autonom um das Timing, während die aktuellen Daten bei jedem Zyklus frisch berechnet werden. Das Ergebnis kann im PCAN-Explorer als Wellenform visualisiert werden (Byte 0 der Nachricht).

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

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

2.2.1.325 Uebung_126b2: Übung zu ISOBUS Send Message Cyclic (mit CB) SINUS-Funktion Plotten



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung demonstriert das zyklische Senden einer ISOBUS-Nachricht, deren Dateninhalt dynamisch zur Laufzeit generiert wird. Konkret wird eine Sinus-Funktion erzeugt, deren Werte in eine CAN-Nachricht verpackt und über das Netzwerk gesendet werden. Dies eignet sich beispielsweise, um Signale im „PCAN Explorer“ zu plotten.

Ein besonderes Merkmal dieser Übung ist die Verwendung des **Callback-Mechanismus**. Anstatt die Daten statisch bereitzustellen, fordert der Sendebaustein (`AlPgnTxNew8Bcycl_REQ`) über eine Adapter-Verbindung (`CB`) neue Daten an, kurz bevor das Paket gesendet wird.

Verwendete Funktionsbausteine (FBs)

In der Hauptanwendung werden folgende Bausteine verwendet, um die Netzwerkkommunikation zu initiieren und den Sendevorgang zu steuern:

- **NmGetCfInfo_1** (`isobus::pgn::NmGetCfInfo`):
  - Dieser Baustein ermittelt notwendige Netzwerkinformationen (z.B. Source Address) für einen bestimmten Knoten (`NODE1`).
  - Er filtert auf bestimmte Parameter (`PEAK_ADD`, `PEAK_FLT`).
- **AlPgnTxNew8Bcycl_REQ** (`isobus::pgn::tx::AlPgnTxNew8Bcycl_REQ`):
  - Dieser Baustein ist für das zyklische Senden der Nachricht verantwortlich.
  - **Parameter**:
    - `u32Pgn`: 61184 (Die Ziel-PGN).

- u16DaSize: 8 (Datenlänge in Bytes).
- u8Priority: 3 (Priorität der Nachricht).
- u16DefRepRate: 500 (Wiederholrate in Millisekunden).

Sub-Bausteine: DataSupply (Uebung_126b2_sub)

Die eigentliche Datenerzeugung findet in einem gekapselten Sub-Baustein statt.

- **Typ**: Uebung_126b2_sub
- **Beschreibung**: Erzeugt eine Sinus-Funktion und stellt diese über ein Callback-Interface bereit.
- **Verwendete interne FBs**:
  - **GEN_SIN**: OSCAT::Basic::POUs::Engineering::signal_generators::GEN_SIN
    - Dient zur Erzeugung des Sinus-Signals.
    - Parameter:
      - PT (Periodendauer) = T#10s
      - AM (Amplitude) = 10.0
      - OS (Offset) = 5.0
  - **F_REAL_TO_DWORD**: iec61131::conversion::F_REAL_TO_DWORD
    - Konvertiert den REAL-Wert des Sinus-Generators in ein DWORD.
  - **BYTES_TO_ARR08B**: logiBUS::utils::conversion::arr::reversing::DWORDS_TO_ARR08B
    - Wandelt das DWORD Format in ein Byte-Array um, das für die CAN-Nachricht benötigt wird.
  - **STRUCT_MUX**: eclipse4diac::convert::STRUCT_MUX
    - Erstellt die Struktur isobus::pgn::CAN_MSG aus den konvertierten Daten.
  - **CallbackFB**: isobus::pgn::tx::CallbackFB
    - Stellt die Verbindung zum Adapter PLUG1 her und triggert die Berechnung, wenn der Sende-Baustein Daten anfordert.
- **Funktionsweise**: Sobald der Sende-Baustein im Hauptnetzwerk bereit ist zu senden, triggert er über den Adapter den CallbackFB. Dieser fordert (REQ) beim GEN_SIN den nächsten Wert an. Der berechnete Sinus-Wert wird konvertiert, in ein Byte-Array zerlegt und in eine CAN-Nachrichtenstruktur verpackt. Diese Struktur wird über den Adapter zurück an den Sende-Baustein gegeben.

Programmablauf und Verbindungen

1. **Initialisierung**:
   - Zuerst wird der Baustein NmGetCfInfo_1 ausgeführt, um die Netzwerkkonfiguration für NODE1 zu laden.
   - Sobald die Informationen verfügbar sind (IND-Event), wird der zyklische Sender AlPgnTxNew8Bcycl_REQ über den Eingang install initialisiert.
2. **Zyklisches Senden**:
   - Der AlPgnTxNew8Bcycl_REQ ist auf eine Zykluszeit von **500 ms** eingestellt.
   - Alle 500 ms wird der Sendevorgang angestoßen.
3. **Datenerzeugung (Callback)**:
   - Der Sender ist über eine Adapter-Verbindung (CB <-> PLUG1) mit dem Sub-Baustein DataSupply verbunden.
   - Vor dem Senden ruft der Sender den Sub-Baustein auf.
   - Im Sub-Baustein wird der aktuelle Wert der Sinus-Kurve berechnet (Periode 10s, Amplitude 10, Offset 5).
   - Der Wert wird in das passende Datenformat (Array of Byte) gewandelt und zurückgegeben.
4. **Ausgabe**:
   - Die PGN 61184 wird mit den aktuellen Sinus-Daten auf den Bus geschrieben. Externe Tools (wie PCAN Explorer) können diese Daten visualisieren.

Zusammenfassung

Diese Übung vermittelt, wie man in 4diac ISOBUS-Anwendungen erstellt, die Daten nicht nur statisch, sondern dynamisch zur Laufzeit berechnen. Durch die Nutzung des Callback-Musters wird eine effiziente Trennung zwischen dem Kommunikationsmanagement (zyklisches Senden) und der Anwendungslogik (Signalgenerierung) erreicht. Das Ergebnis ist eine auf dem CAN-Bus sichtbare Sinus-Schwingung.

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

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

2.2.1.326 Uebung_126b2_sub

Hinweis

Diese Übung existiert nicht mehr als eigene Übung. Der Baustein wurde in die Bibliothek verschoben und heißt jetzt SIN_TO_PCAN_Callback_DWord. Beschreibung, Schnittstelle und Netzwerk stehen dort.

2.2.1.327 Uebung_126b_sub

Hinweis

Diese Übung existiert nicht mehr als eigene Übung. Der Baustein wurde in die Bibliothek verschoben und heißt jetzt SIN_TO_PCAN_Callback_Byte. Beschreibung, Schnittstelle und Netzwerk stehen dort.

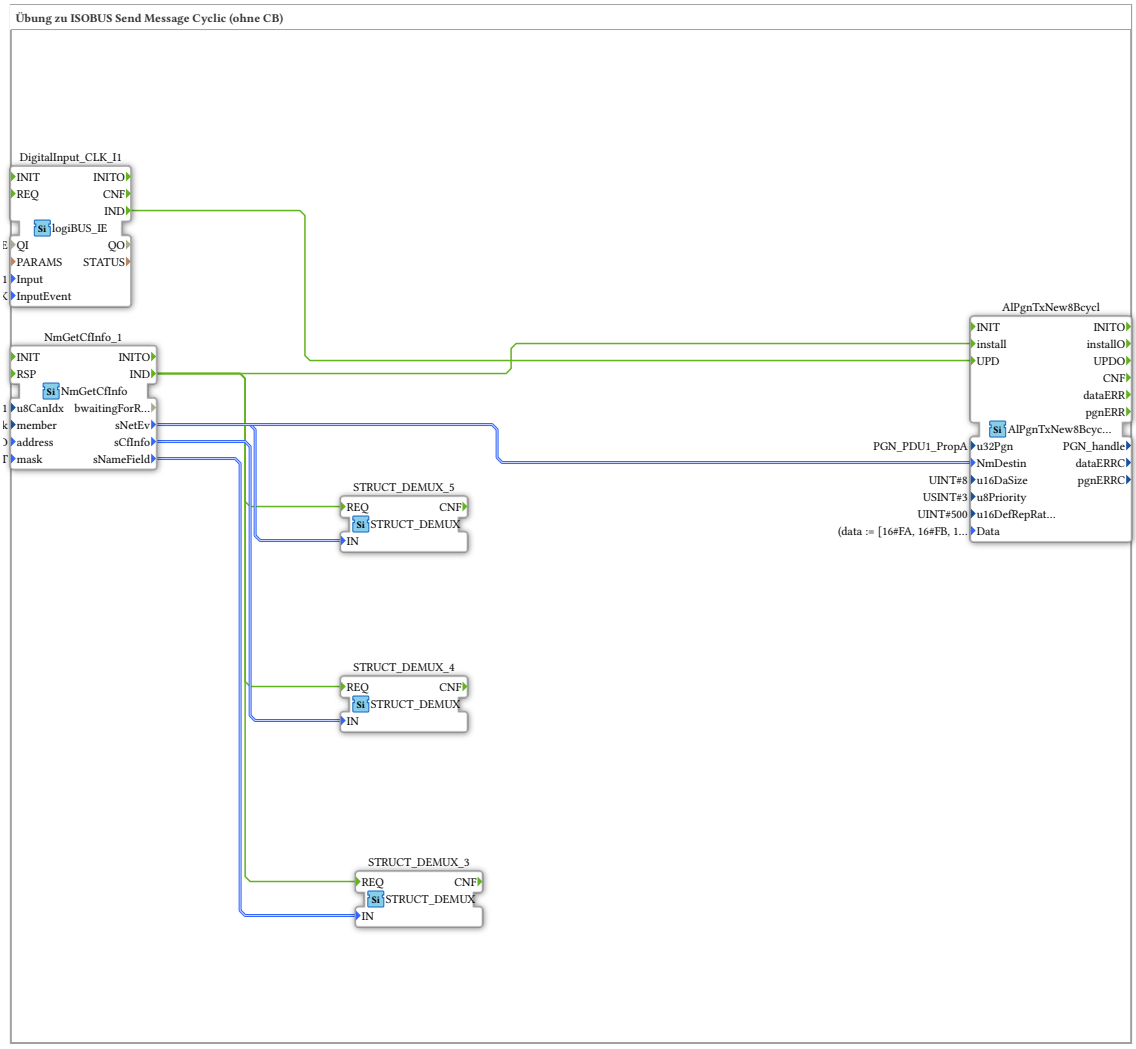
2.2.1.328 Uebung_127: Übung zu ISOBUS Send Message Cyclic (ohne CB)



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_127`.

Übersicht

Variante des zyklischen Sendens unter Verwendung von `AlPgnTxNew8Bcycl` ohne Callback. In dieser Übung werden die zu sendenden Daten fest als Parameter am Baustein hinterlegt. Ein Ereignis am Eingang `UPD` (Update) ermöglicht es der Applikation jedoch, den Inhalt der Nachricht bei Bedarf zu ändern. Dies ist eine einfachere Alternative zum Callback, wenn sich die Daten nicht in jedem Zyklus ändern.



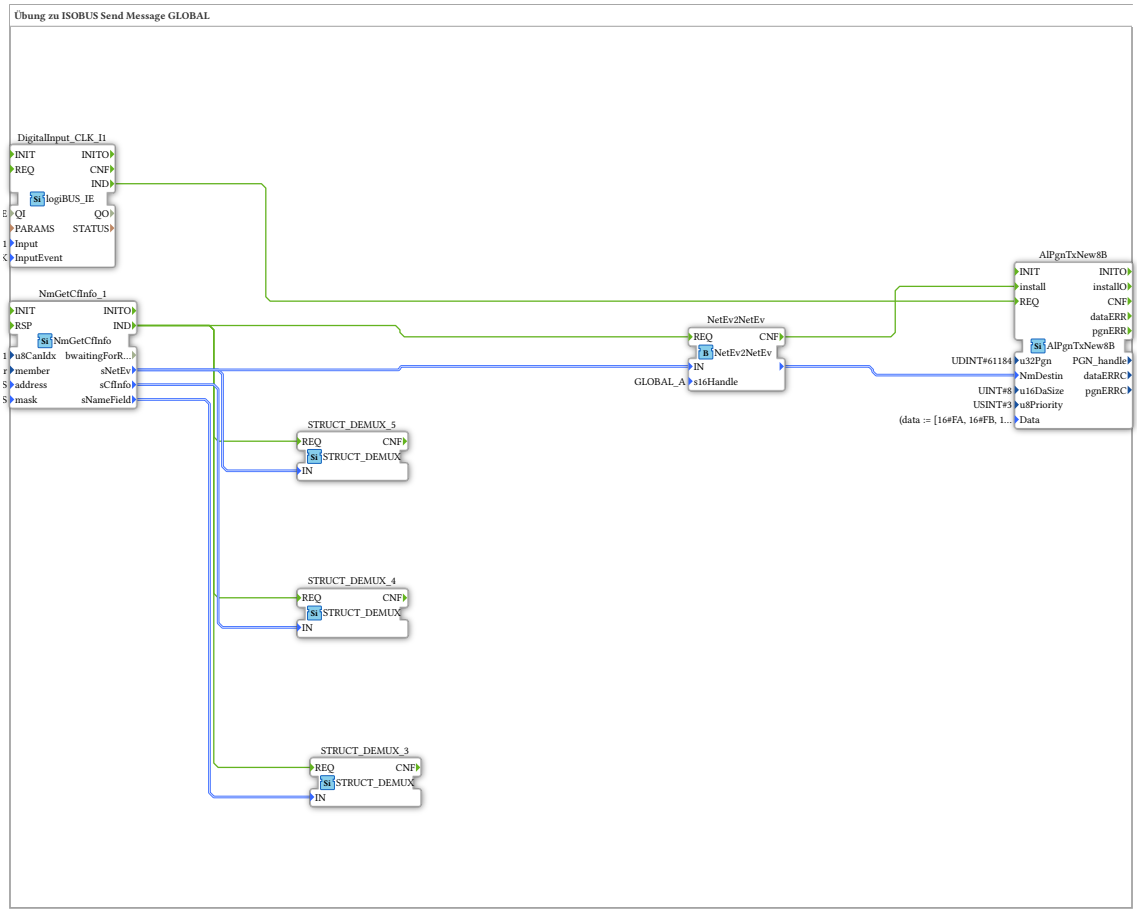
2.2.1.329 Uebung_128: Übung zu ISOBUS Send Message GLOBAL



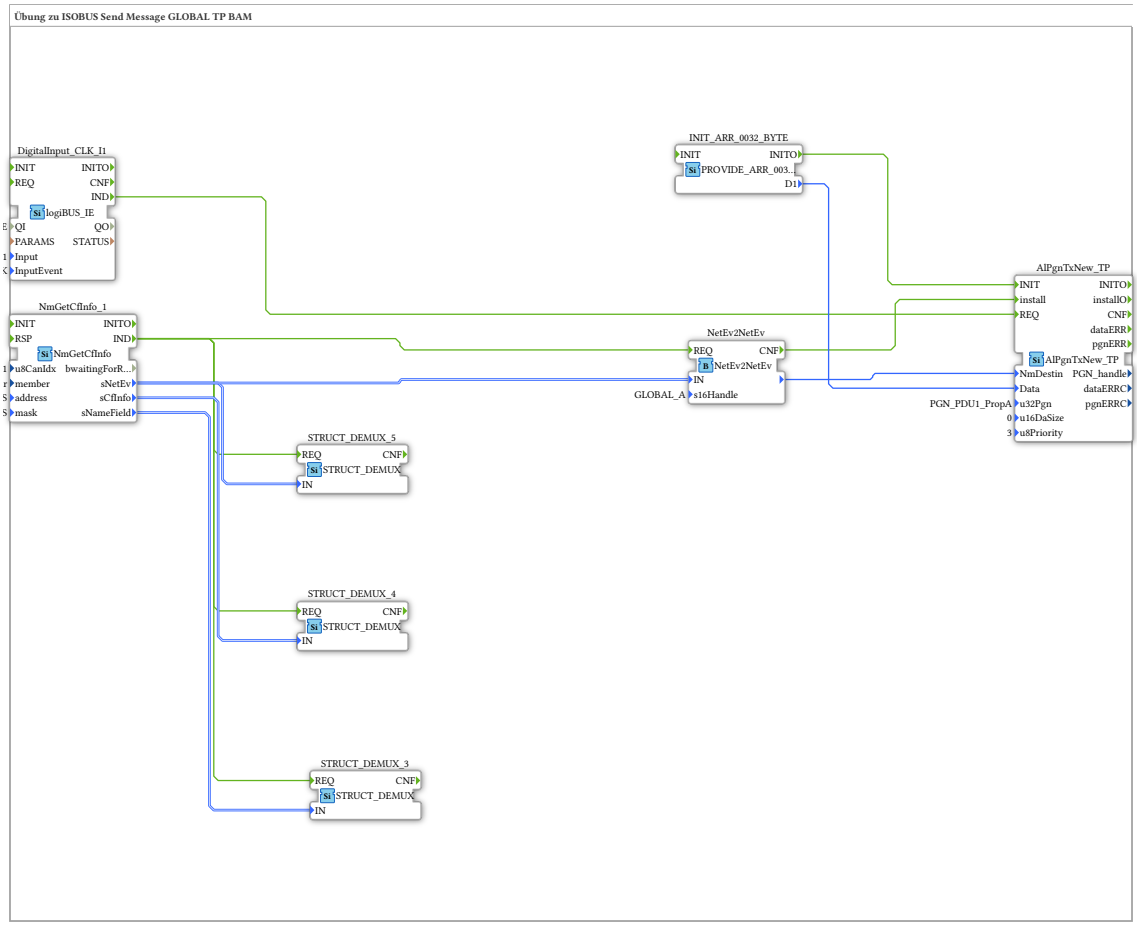
Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_128`.

Übersicht

In dieser Übung wird eine Nachricht an alle Teilnehmer im Netzwerk gleichzeitig gesendet Broadcast. Dazu wird der Baustein `NetEv2NetEv` mit dem speziellen Handle `GLOBAL_A` (Adresse 255) genutzt. Die resultierende Netzwerk-Identität wird als Ziel (`NmDestin`) für den Sende-Baustein verwendet. Eine so gesendete Nachricht wird von jedem Gerät am ISOBUS empfangen und kann für allgemeine Informationen oder Synchronisations-Signale genutzt werden.


Übung zu ISOBUS Send Message GLOBAL
DigitalInput_CLK_I1
INIT
REQ
INITO
CNF
IND
St logiBUS_IE
QI
PARAMS
Input
InputEvent
QO
STATUS
NmGetCfInfo_1
INIT
RSP
St NmGetCfInfo
u8CanIdx
member
address
mask
INITO
IND
bwaitingForR...
sNetEv
sCfInfo
sNameField
NetEv2NetEv
REQ
B NetEv2NetEv
IN
s16Handle
CNF
GLOBAL_A
AlPgnTxNew8B
INIT
install
REQ
INITO
installO
CNF
dataERR
pgnERR
St AlPgnTxNew8B
u32Pgn
NmDestin
u16DaSize
u8Priority
Data
PGN_handle
dataERRC
pgnERRC
UDINT#61184
UINT#8
USINT#3
(data := [16#FA, 16#FB, 1...
STRUCT_DEMUX_5
REQ
St STRUCT_DEMUX
IN
CNF
STRUCT_DEMUX_4
REQ
St STRUCT_DEMUX
IN
CNF
STRUCT_DEMUX_3
REQ
St STRUCT_DEMUX
IN
CNF

2.2.1.330 Uebung_128b: Übung zu ISOBUS Send Message GLOBAL TP BAM



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert das Senden einer ISOBUS-Nachricht unter Verwendung des **Transportprotokolls (TP)** mit der Methode **Broadcast Announce Message (BAM)**. Dabei wird eine Nachricht an die globale Adresse (Broadcast) gesendet. Da die Datenmenge 8 Bytes überschreitet (hier 32 Bytes), ist die Verwendung des Transportprotokolls notwendig.

Verwendete Funktionsbausteine (FBs)

In dieser Übung werden verschiedene Funktionsbausteine verschaltet, um die Netzwerkkommunikation und Datenerzeugung zu realisieren.

Hauptkomponenten

- **isobus::pgn::NmGetCfInfo** (`NmGetCfInfo_1`)
  - Dieser Baustein ruft Informationen über eine Control Function (CF) im Netzwerk ab.
  - **Parameter**:
    - `u8CanIdx`: `NODE1` (CAN-Knoten 1)
    - `member`: `thisMember` (Bezieht sich auf den eigenen Teilnehmer)
    - `address`, `mask`: `FLT_ALL_PASS` (Filtereinstellungen)
  - **Funktion**: Er stellt Netzwerkereignisse (`sNetEv`) und Namensinformationen bereit, die für die Initialisierung des Sende-Bausteins benötigt werden.
- **isobus::pgn::NetEv2NetEv** (`NetEv2NetEv`)
  - Ein Konverter-Baustein, der Netzwerkereignisse verarbeitet und Zieladressen zuweist.
  - **Parameter**:
    - `s16Handle`: `GLOBAL_A` (Definiert das Ziel als globale Adresse/Broadcast).

  - **Funktion**: Er wandelt das Netzwerkereignis des eigenen Geräts in ein Ereignis um, das für eine globale Übertragung (Broadcast) konfiguriert ist.

- **isobus::pgn::tx::AlPgnTxNew_TP** (`AlPgnTxNew_TP`)
  - Der eigentliche Sende-Baustein für PGNs unter Verwendung des Transportprotokolls (TP).
  - **Parameter**:
    - `u32Pgn`: `61184` (Proprietary A PGN).
    - `u16DaSize`: `0` (Wird dynamisch überschrieben).
    - `u8Priority`: `3`.
  - **Eingänge**:
    - `NmDestin`: Erhält die Zielinformation (Global) vom `NetEv2NetEv`.
    - `Data`: Erhält die 32-Byte-Nutzdaten.
    - `install`: Initialisiert den Sende-Handle.
    - `REQ`: Löst das Senden aus.

- **logiBUS::io::DI::logiBUS_IE** (`DigitalInput_CLK_I1`)
  - Verarbeitet digitale Eingabesignale.
  - **Parameter**:
    - `Input`: `Input_I1`
    - `InputEvent`: `BUTTON_SINGLE_CLICK`
  - **Funktion**: Dient als Auslöser (Trigger) für den Sendevorgang.

- **eclipse4diac::convert::providers::PROVIDE_ARR_0032_BYTE** (`INIT_ARR_0032_BYTE`)
  - Erzeugt ein statisches Byte-Array.
  - **Parameter**:
    - `D1`: Ein Array mit 32 Bytes (beginnend mit `16#01, 16#00... 16#AA...`).
  - **Funktion**: Stellt die Nutzdaten (Payload) für die ISOBUS-Nachricht bereit.

Debugging / Visualisierung

Folgende Bausteine dienen der Aufschlüsselung von Strukturen zu Diagnosezwecken:

- **eclipse4diac::convert::STRUCT_DEMUX** (`STRUCT_DEMUX_3`): Zerlegt `isobus::pgn::NAMEFIELD_T`.
- **eclipse4diac::convert::STRUCT_DEMUX** (`STRUCT_DEMUX_4`): Zerlegt `isobus::pgn::CF_INFO_T`.
- **eclipse4diac::convert::STRUCT_DEMUX** (`STRUCT_DEMUX_5`): Zerlegt `isobus::pgn::ISONETEVENT_T`.

Programmablauf und Verbindungen

Der Ablauf der Übung gestaltet sich wie folgt:

1. **Initialisierung**:
   - Der Baustein `NmGetCfInfo_1` liefert Informationen über den eigenen Netzwerkknoten. Das `IND`-Ereignis triggert die nachfolgenden Schritte.
   - Die Netzwerkinformationen (`sNetEv`) werden an `NetEv2NetEv` geleitet.
   - Gleichzeitig stellt `INIT_ARR_0032_BYTE` ein 32-Byte-Datenpaket zur Verfügung und initialisiert den Dateneingang von `AlPgnTxNew_TP`.
2. **Konfiguration des Senders**:
   - Der Baustein `NetEv2NetEv` ist mit dem Handle `GLOBAL_A` konfiguriert. Das bedeutet, er bereitet den Sende-Baustein darauf vor, an die globale Adresse (255) zu senden.
   - Das Ergebnis von `NetEv2NetEv` wird an den Eingang `NmDestin` des `AlPgnTxNew_TP` gelegt und über das Event `install` bestätigt. Damit weiß der Sender, dass er ein Broadcast-Telegramm senden soll.
3. **Sendevorgang (TP BAM)**:
   - Durch Betätigen des Tasters `Input_I1` (Single Click) am Baustein `DigitalInput_CLK_I1` wird das Event `REQ` am Sende-Baustein `AlPgnTxNew_TP` ausgelöst.
   - Da die Datenlänge (32 Byte) größer als 8 Byte ist und das Ziel die globale Adresse ist, verwendet der Baustein automatisch das **BAM-Protokoll** (Broadcast Announce Message), um die Daten segmentiert zu übertragen.
   - Gesendet wird die PGN 61184 (Proprietary A) mit Priorität 3.

Zusammenfassung

In dieser Übung wird die Handhabung von ISOBUS-Transportprotokollen vertieft. Im Speziellen wird gezeigt, wie größere Datenmengen (> 8 Byte) mittels `AlPgnTxNew_TP` an alle Teilnehmer im Netzwerk (Broadcast) gesendet werden. Die Kombination aus der PGN-Konfiguration, der Datenquelle (`INIT_ARR`) und der Adressierung (`GLOBAL_A`) führt zur automatischen Aushandlung einer BAM-Übertragung.

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

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

2.2.1.331 Uebung_12x_sub

Hinweis

Diese Übung existiert in den aktuellen Trainingsquellen nicht mehr (entfernt, zusammengelegt oder umbenannt). Diese Seite bleibt nur als Platzhalter für bestehende Verweise erhalten. Die aktuellen Übungen stehen in der Übungsübersicht.

2.2.1.332 Uebung_130: Übung zu ISOBUS Receive Message



Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_130`. Hier wird das Gegenstück zum Senden gezeigt: Das gezielte Empfangen von herstellerspezifischen Nachrichten.

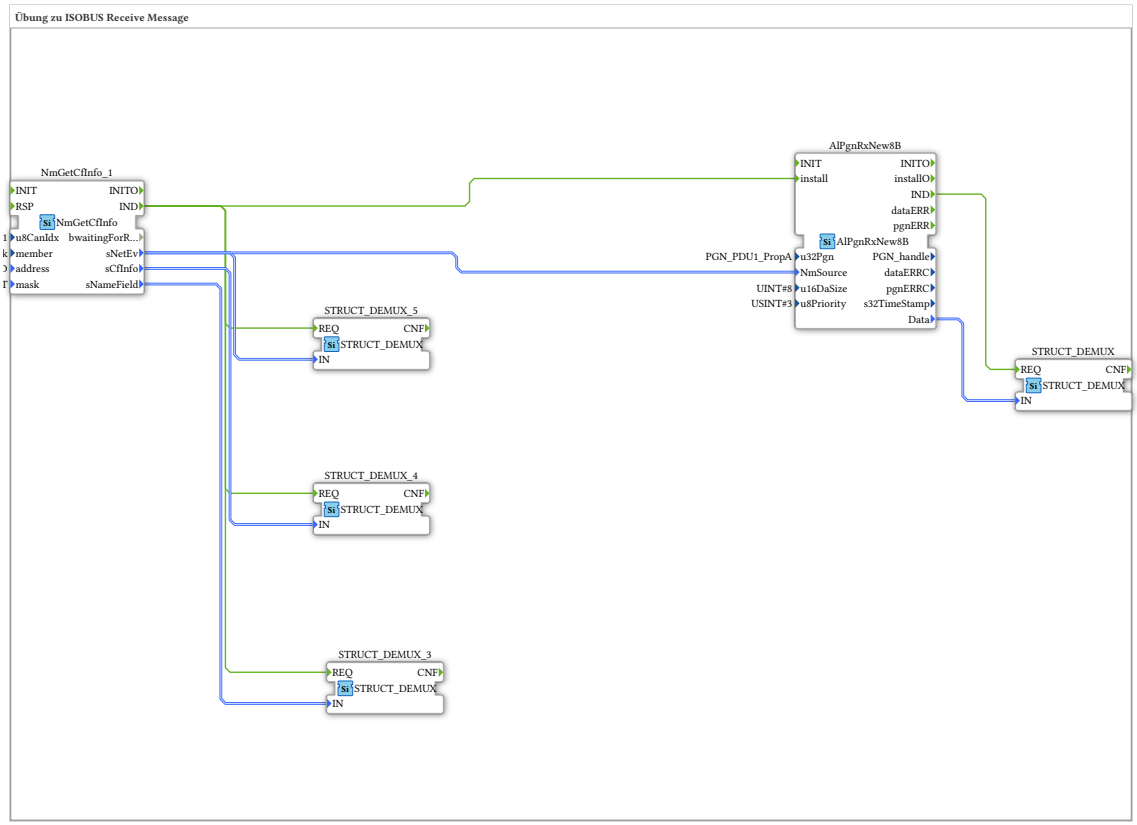
Ziel der Übung

Verwendung des Bausteins `AlPgnRxNew8B`. Es wird demonstriert, wie man auf eine spezifische Nachricht (PGN) eines bestimmten Partners lauscht und den Inhalt für das eigene Programm auswertet.

Beschreibung und Komponenten

In `Uebung_130.SUB` wird ein Empfangs-Filter für eine herstellerspezifische PGN konfiguriert.

Funktionsbausteine (FBs)



- **`NmGetCfInfo_1`**: Identifiziert den Absender der Nachricht (Source).
- **`AlPgnRxNew8B`**: Der Empfangs-Baustein. Er filtert alle CAN-Nachrichten und lässt nur die passende PGN durch.
- **`STRUCT_DEMUX`**: Zerlegt die empfangene 8-Byte Nachricht wieder in einzelne Signale.

Funktionsweise

1. Der Baustein wird über `install` im System registriert und mit dem gewünschten Absender (`NmSource`) verknüpft.
2. Sobald der Partner die passende Nachricht (PGN `61184`) sendet, erkennt dies der Baustein.
3. Er feuert das Ereignis `IND` (Indication) ab und stellt das Datenpaket am Port `Data` bereit.
4. Über den Demultiplexer kann das Programm nun auf den Inhalt der Nachricht reagieren.

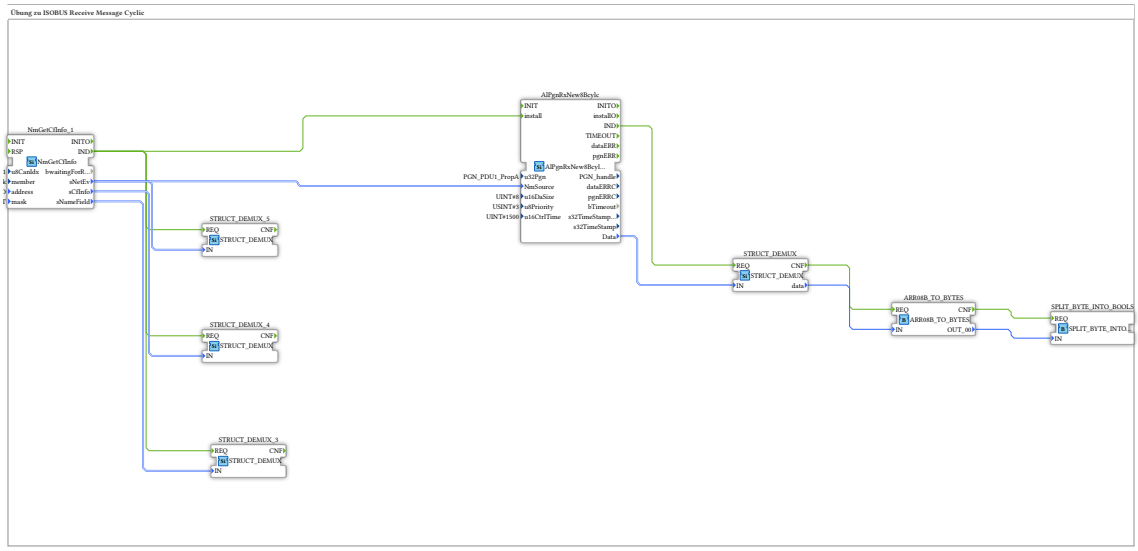
Dies ermöglicht eine private Kommunikation zwischen zwei spezifischen Geräten am Bus, ohne andere Teilnehmer zu stören.

2.2.1.333 Uebung_131: Übung zu ISOBUS Receive Message Cyclic

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_131`. Hier wird der Empfang von Nachrichten um eine Zeit-Überwachung (Timeout) ergänzt.

Podcast



- Die drei Timer der DIN EN 61131-3 entschlüsselt – TP, TON & TOF präzise erklärt
- DIN EN 61131-3 vs. 61499-1: Dein Wegweiser durch die Normen der Industrieautomatisierung
- DIN EN 61131-3: Das Herz der Land- und Baumaschinen-Mechatronik und der Sprung in die Zukunft mit Ob
- FB_TOF und E_TOF: Verzögerungstimer in IEC 61131-3 und 61499
- IEC 61499 vs. 61131: Brauchen wir einen neuen Standard für IIoT? Analyse einer hitzigen Debatte um Verteilte Intelligenz

---

Übersicht

Verwendung des Bausteins `AlPgnRxNew8Bcylc`. Dieser Baustein ist speziell für Nachrichten gedacht, die regelmäßig (zyklisch) erwartet werden. Über den Parameter `u16CtrlTime = 1500`ms wird eine Kontrollzeit definiert. Sollte der Partner für länger als 1,5 Sekunden keine Nachricht mehr senden, wird dies als Kommunikationsabbruch gewertet. Die Applikation kann auf diesen Fehlerfall reagieren, um die Maschine in einen sicheren Zustand zu bringen.

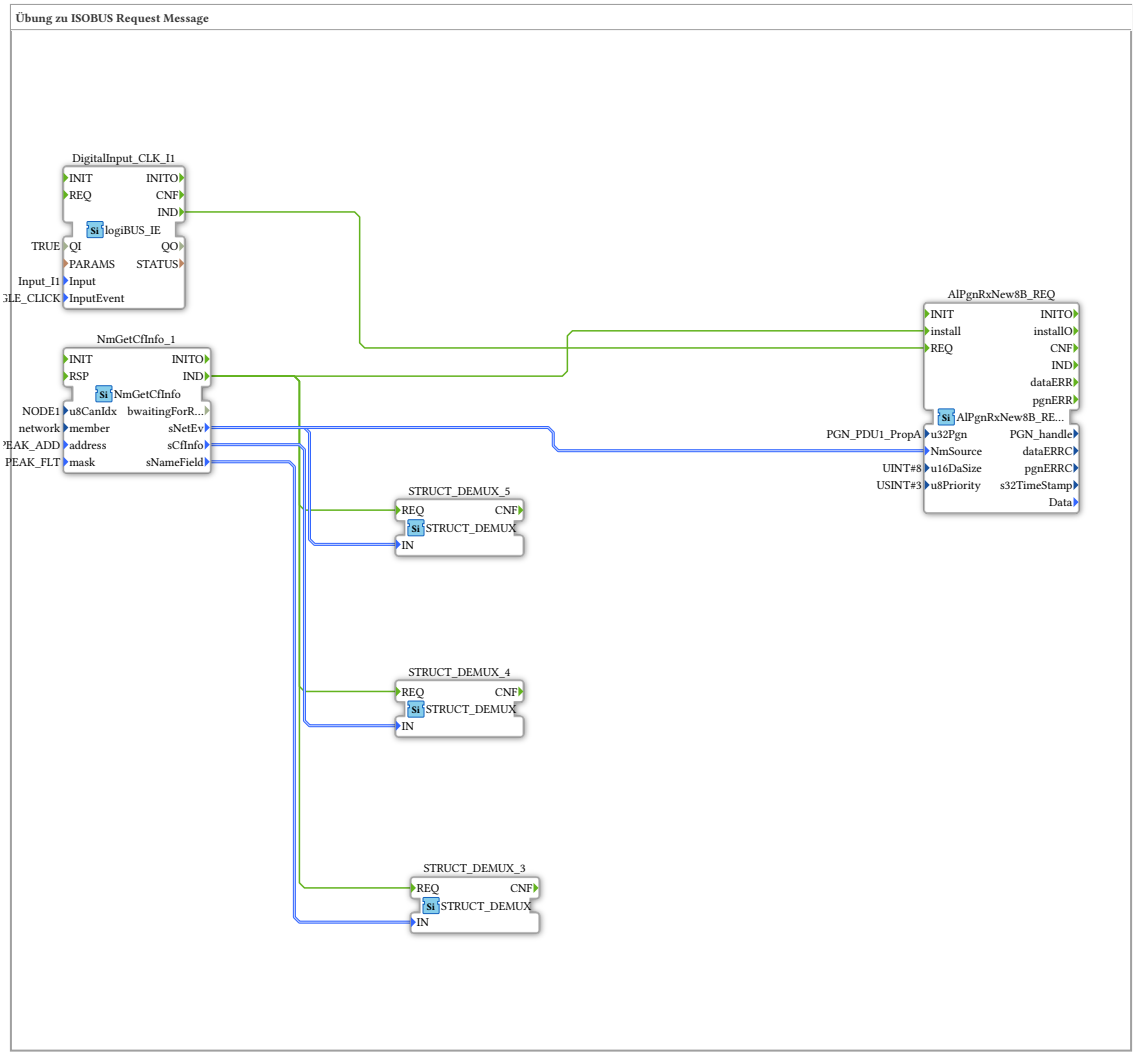
2.2.1.334 Uebung_132: Übung zu ISOBUS Request Message

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_132`.

Übersicht

In dieser Übung wird der Baustein `AlPgnRxNew8B_REQ` verwendet.



Anstatt passiv auf eine Nachricht zu warten, kann die Steuerung hier aktiv eine Anfrage (Request) nach einer bestimmten PGN an den Partner senden. Ein Klick auf Taster **I1** löst das `REQ`-Ereignis aus, woraufhin die Steuerung die entsprechende Anforderungs-Nachricht auf den Bus schickt. Sobald der Partner antwortet, wird dies als regulärer Empfang (`IND`) gewertet und verarbeitet.

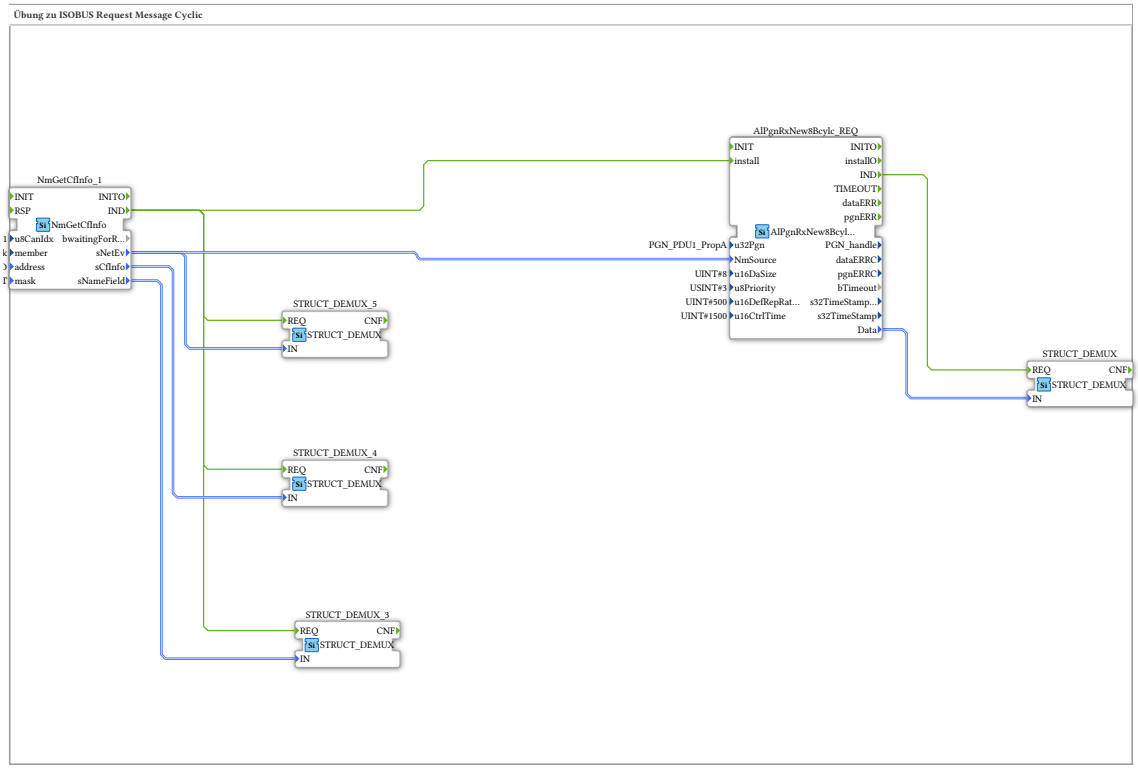
2.2.1.335 Uebung_133: Übung zu ISOBUS Request Message Cyclic

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_133`.

Übersicht

Kombination aus Abfrage und zyklischer Überwachung unter Verwendung von `AlPgnRxNew8Bcylc_REQ`. Dieser Baustein sendet in festen Abständen eigenständig Anfragen an den Partner und überwacht gleichzeitig das Eintreffen der Antworten innerhalb der Kontrollzeit. Dies ist die sicherste Form der Kommunikation für Daten, die nicht automatisch vom Partner gesendet werden, aber dennoch permanent aktuell sein müssen.



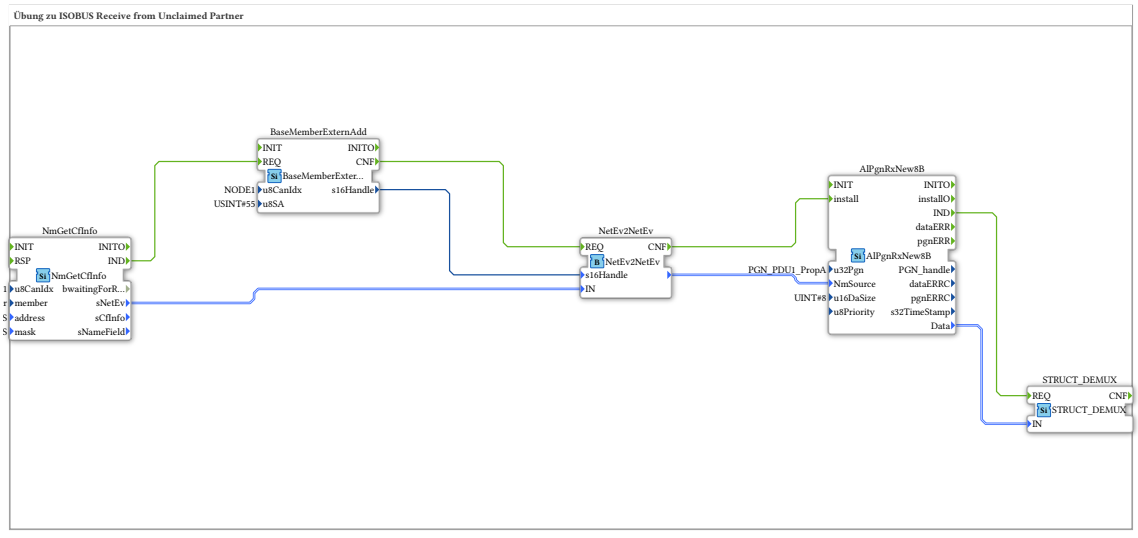
2.2.1.336 Uebung_134: Übung zu ISOBUS Receive from Unclaimed Partner

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_134`.

Übersicht

In dieser Übung wird eine Lösung für die Kommunikation mit Geräten gezeigt, die kein standardkonformes ISOBUS-Management (Address Claiming) durchführen. Unter Verwendung des Bausteins `BaseMemberExternAdd` wird manuell ein Kommunikations-Handle für eine feste Quelladresse (hier `u8SA = 55`) erstellt. Dieses Handle wird genutzt, um Nachrichten von einem „Unclaimed Partner“ zu empfangen, der seine Identität nicht über die Namensverwaltung preisgibt. Dies ist oft bei der Integration von einfachen Sensoren oder Altgeräten notwendig.



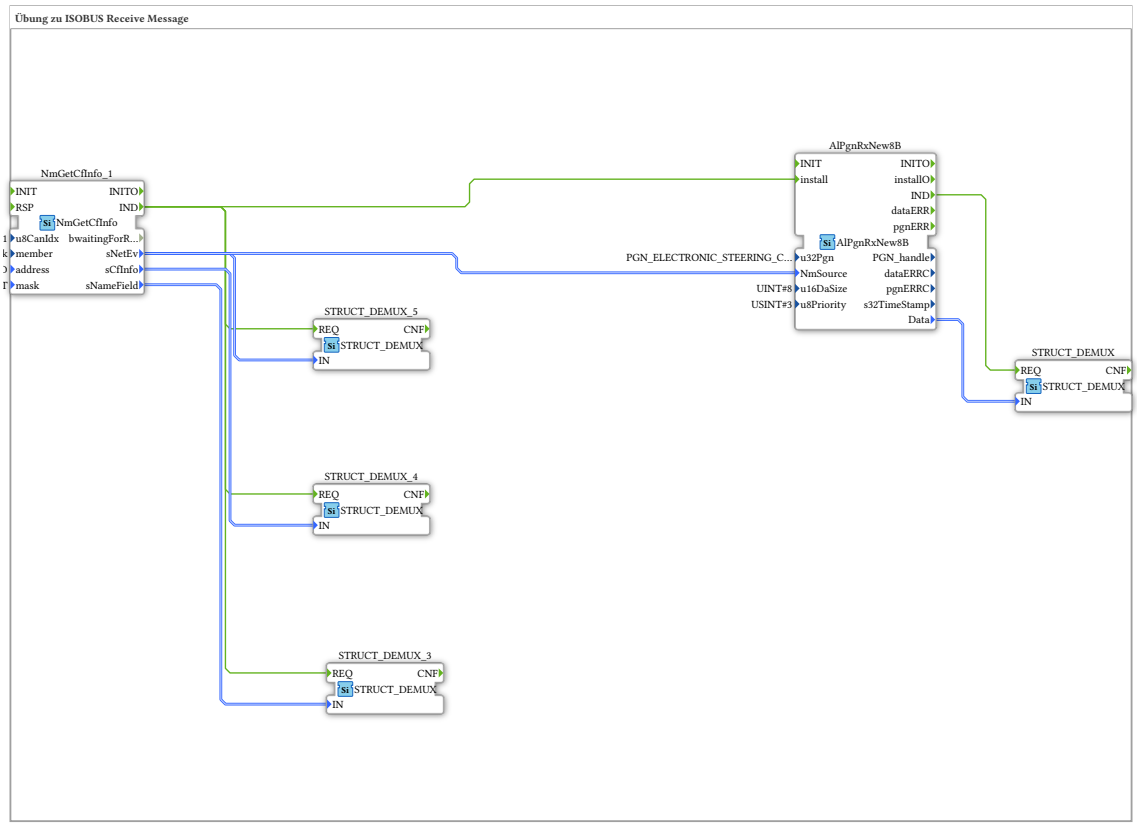
2.2.1.337 Uebung_135: Übung zu ISOBUS Receive Message

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_135`. Wir empfangen eine ISOBUS-Nachricht (PGN) und zerlegen die enthaltenen Strukturdaten in ihre Einzelfelder.

Ziel der Übung

Anmeldung eines Steuergeräts (CF) auf dem ISOBUS-Netzwerk verfolgen, dessen zugehörige Empfangs-PGN abonnieren und die empfangenen Rohdaten (`8B`, 8 Byte) in einzelne Felder zerlegen.

Beschreibung und Komponenten



- **`NmGetCfInfo_1`**: `isobus::pgn::NmGetCfInfo`, verfolgt die Netzwerk-Anmeldung eines Steuergeräts (`address = PRIM_TECU_ADD`, `mask = PRIM_TECU_FLT`) und liefert dessen Namens-, Kennungs- und Netzwerkereignis-Strukturen.
- **`AlPgnRxNew8B`**: `isobus::pgn::rx::AlPgnRxNew8B`, abonniert eine konkrete PGN (`u32Pgn = PGN_ELECTRONIC_STEERING_CONTROL`) für das erkannte Steuergerät und empfängt deren 8-Byte-Rohdaten.
- **`STRUCT_DEMUX`** (mehrfach instanziiert): `eclipse4diac::convert::STRUCT_DEMUX`, zerlegt eine Struktur in ihre einzelnen Felder.

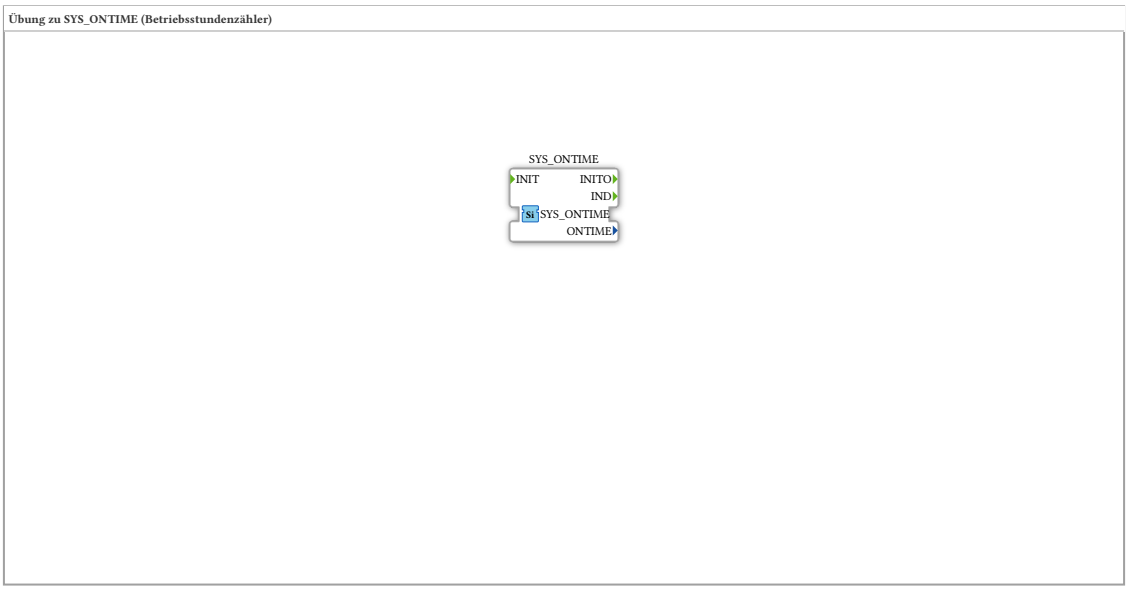
Funktionsweise

1. `NmGetCfInfo_1` meldet (`IND`), sobald das gesuchte Steuergerät im Netzwerk erkannt wurde.
2. Die gelieferten Strukturen (Name, Kennung, Netzwerkereignis) werden über drei `STRUCT_DEMUX`-Instanzen in ihre Einzelfelder zerlegt.
3. Parallel installiert `NmGetCfInfo_1.IND` den PGN-Empfang `AlPgnRxNew8B` für dieses Steuergerät.
4. Empfängt `AlPgnRxNew8B` eine neue Nachricht (`IND`), wird deren 8-Byte-Rohdatum ebenfalls per `STRUCT_DEMUX` in seine Einzelfelder zerlegt.

2.2.1.338 Uebung_140: Übung zu SYS_ONTIME (Betriebsstundenzähler)

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_140`. Hier wird die Erfassung der Systemlaufzeit demonstriert.

🎧 Podcast



- Von 1400 Fehlern zum sauberen Code: Die Migration der „Getreidehacke“ auf Eclipse 4diac™ 3.0 und die Macht der AX-Adapter

---

Ziel der Übung

Verwendung des Bausteins `SYS_ONTIME`. Ziel ist es, die kumulierte Zeit zu erfassen, in der die Steuerung eingeschaltet und aktiv ist.

Beschreibung und Komponenten

Die Subapplikation `Uebung_140.SUB` nutzt einen speziellen Messbaustein zur Zeitüberwachung.

Funktionsbausteine (FBs)

- **`SYS_ONTIME`**: Typ `logiBUS::signalprocessing::measurement::SYS_ONTIME`. Dieser Baustein misst die Zeit seit dem letzten Systemstart oder die kumulierte Gesamtzeit je nach Implementierung.

Funktionsweise

Der Baustein läuft im Hintergrund mit. Er bietet typischerweise Ausgänge für Sekunden, Minuten und Stunden. Diese Daten können dann dauerhaft gespeichert (NVS) oder auf dem Service-Menü des Terminals angezeigt werden.

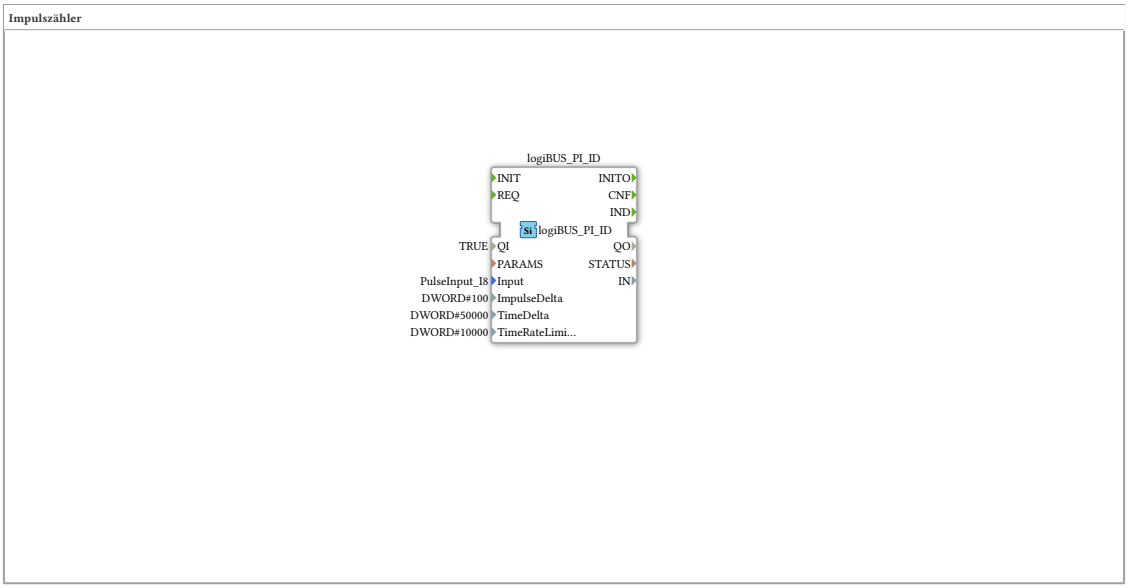
Anwendungsbeispiel

Wartungsintervalle: Die Steuerung zählt die Betriebsstunden der Maschine. Sobald ein Grenzwert (z.B. 500 Stunden) erreicht ist, wird dem Bediener am Terminal eine Meldung angezeigt: „Ölwechsel erforderlich“. Dies garantiert die Einhaltung von Wartungsplänen und erhöht die Lebensdauer der Maschine.

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

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

2.2.1.339 Uebung_150: Impulszähler



• – ◦ • – ◦ • – ◦ •

Einleitung

Die Übung **Uebung_150** implementiert eine Sub-Applikation (SubApp), die als „Impulszähler“ fungiert. Sie dient der Konfiguration und Einbindung eines hardwareseitigen Pulse-Inputs über das logiBUS-System. Ziel ist es, eine Schnittstelle zu einem physikalischen Eingang (`PulseInput_I8`) herzustellen und Parameter für die Erfassung festzulegen.

Verwendete Funktionsbausteine (FBs)

In dieser SubApp wird ein spezifischer Funktionsbaustein verwendet, um die Kommunikation mit der Hardware zu konfigurieren.

Sub-Bausteine: logiBUS_PI_ID

Dieser Baustein ist die zentrale Komponente der Übung und stellt die Verbindung zur Hardware-Eingangsschicht her.

- **Typ**: `logiBUS::io::PI::logiBUS_PI_ID`
- **Verwendete interne FBs**:
  - **Bausteinname**: `logiBUS_PI_ID`
    - **Typ**: `logiBUS::io::PI::logiBUS_PI_ID`
    - **Parameter**:
      - `QI` = `TRUE` (Aktivierung des Bausteins/Initialisierung)
      - `Input` = `PulseInput_I8` (Zuweisung des spezifischen Hardware-Eingangs)
      - `ImpulseDelta` = `100` (Schwellenwert für Impulsänderungen)
      - `TimeDelta` = `50000` (Zeitintervall für die Aktualisierung/Messung)
    - **Ereignisausgang/-eingang**: *Keine expliziten Verbindungen im XML definiert.*
    - **Datenausgang/-eingang**: *Keine expliziten Verbindungen im XML definiert.*
- **Funktionsweise**: Der Baustein `logiBUS_PI_ID` initialisiert einen Pulse-Input-Kanal. Durch das Setzen von `QI` auf `TRUE` wird der Baustein aktiviert. Der Parameter `Input` legt fest, dass der physikalische Eingang `PulseInput_I8` verwendet wird. Die Parameter `ImpulseDelta` und `TimeDelta` konfigurieren die Empfindlichkeit und das Zeitverhalten des Zählers, was steuert, wann und wie oft Aktualisierungen über den Bus gesendet oder verarbeitet werden.

Programmablauf und Verbindungen

Da es sich bei dieser Übung um eine reine Konfiguration eines Hardware-Treibers innerhalb einer SubApp handelt, gibt es keinen komplexen Programmablauf oder Verschaltungen zwischen mehreren Bausteinen.

- **Initialisierung**: Der Baustein ist statisch parametriert. Beim Start der Applikation wird der Hardware-Treiber mit den Werten `100` für das Impuls-Delta und `50000` für das Zeit-Delta geladen.
- **Schnittstellen**: Die SubApp `Uebung_150` selbst definiert keine externen Ein- oder Ausgänge in ihrer `SubAppInterfaceList`. Sie wirkt als in sich geschlossenes Konfigurationsmodul für den logiBUS.
- **Lernziele**:
  - Verständnis der Anbindung von Hardware-Inputs über logiBUS.
  - Konfiguration von Zählerparametern (Delta-Werte).
  - Umgang mit SubApp-Typen zur Kapselung von Hardware-Konfigurationen.

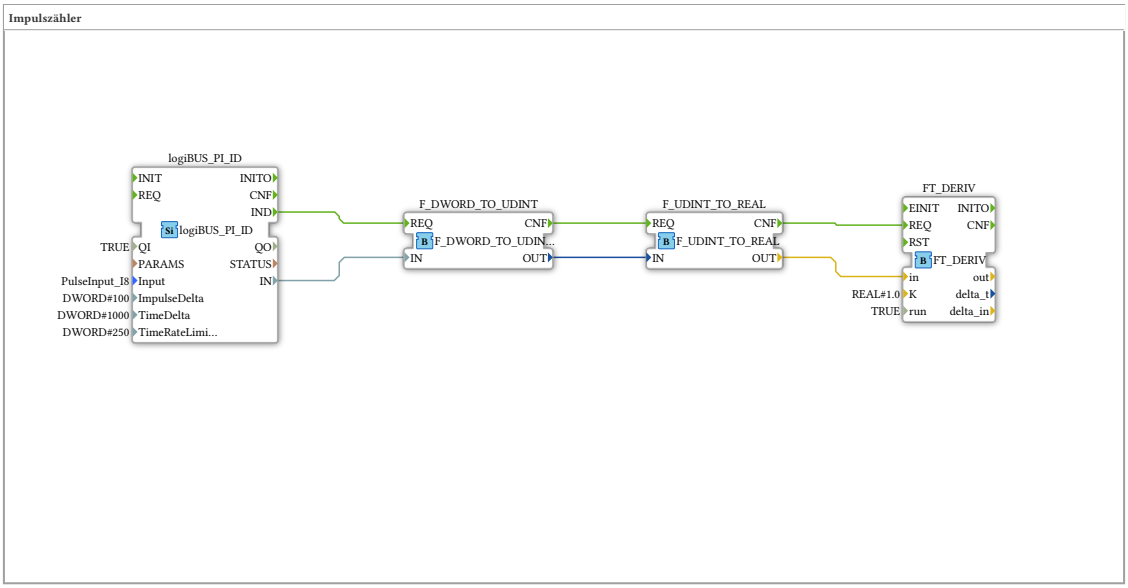
Zusammenfassung

Die Übung **Uebung_150** stellt eine grundlegende Konfiguration für einen Impulszähler dar. Sie nutzt den Baustein `logiBUS_PI_ID`, um den Hardware-Eingang `PulseInput_I8` mit spezifischen Parametern für Impuls- und Zeitintervalle zu initialisieren. Diese Übung ist essenziell für das Verständnis der Hardware-Abstraktionsschicht in 4diac-Systemen, die logiBUS verwenden.

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

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

2.2.1.340 Uebung_151: Impulszähler



(Hier Bild der Übung einfügen, falls vorhanden)

• – ○ • – ○ • – ○ •

Einleitung

Diese Übung implementiert eine Sub-Applikation (`SubAppType`) mit dem Namen **Uebung_151**. Ziel der Übung ist die Erfassung und Verarbeitung von Impulssignalen über eine Hardwareschnittstelle. Die eingehenden Rohdaten werden konvertiert und anschließend wird mittels eines mathematischen Bausteins die Ableitung (Änderungsrate) berechnet. Dies dient typischerweise dazu, aus einem Zählerstand eine Geschwindigkeit oder Frequenz zu ermitteln.

Verwendete Funktionsbausteine (FBs)

In dieser Übung werden verschiedene Standard- und Bibliotheksbausteine innerhalb des Netzwerks verwendet.

Sub-Bausteine: Uebung_151 (Netzwerk-Komponenten)

Hier werden die internen Bausteine beschrieben, die im Netzwerk dieser SubApp verschaltet sind.

- **Verwendete interne FBs**:
  - **logiBUS_PI_ID**: `logiBUS::io::PI::logiBUS_PI_ID`

- Dieser Baustein stellt die Schnittstelle zur Hardware (Pulse Input) dar.
- **Parameter**:
  - `QI` = `TRUE` (Aktiviert den Baustein)
  - `Input` = `PulseInput_I8` (Referenz auf den physikalischen Eingang)
  - `ImpulseDelta` = `100`
  - `TimeDelta` = `1000`
- **Funktionsweise**: Er liefert Prozessdaten basierend auf den konfigurierten Parametern für den Impulseingang.

– **F_DWORD_TO_UDINT**: `iec61131::conversion::F_DWORD_TO_UDINT`

- **Typ**: Konvertierungsbaustein.
- **Funktionsweise**: Wandelt den Datentyp `DWORD` (Doppelwort) in `UDINT` (Unsigned Double Integer) um. Dies ist notwendig, um die Rohdaten des Hardware-Bausteins für die mathematische Weiterverarbeitung vorzubereiten.

– **F_UDINT_TO_REAL**: `iec61131::conversion::F_UDINT_TO_REAL`

- **Typ**: Konvertierungsbaustein.
- **Funktionsweise**: Wandelt den Datentyp `UDINT` in `REAL` (Gleitkommazahl) um. Die `REAL`-Darstellung wird für den nachfolgenden Ableitungsbaustein benötigt.

– **FT_DERIV**: `OSCAT::Basic::POUs::Engineering::Control::FT_DERIV`

- **Typ**: Regelungstechnischer Baustein (Ableitung).
- **Parameter**:
  - `K` = `1.0` (Verstärkungsfaktor)
  - `run` = `TRUE` (Baustein ist aktiv)
- **Funktionsweise**: Berechnet die zeitliche Ableitung des Eingangssignals. In diesem Kontext wird aus der Änderung des Zählerstandes (Impulse) über die Zeit die Frequenz bzw. Geschwindigkeit ermittelt.

Programmablauf und Verbindungen

Der Ablauf der Übung wird durch die Ereignis- und Datenkette definiert:

1. **Signalerfassung**: Der Baustein `logiBUS_PI_ID` erfasst Signale am Eingang `PulseInput_I8`. Sobald neue Daten verfügbar sind, wird das Ereignis `IND` ausgelöst und der Datenwert am Ausgang `IN` bereitgestellt.
2. **Typkonvertierung**:
   - Das Signal gelangt zunächst zum Baustein `F_DWORD_TO_UDINT`.
   - Das Ergebnis wird an `F_UDINT_TO_REAL` weitergeleitet.
   - Diese Kette sorgt dafür, dass das Signal von einem binären Rohformat (`DWORD`) in eine fließkommabasierte Zahl (`REAL`) gewandelt wird.
3. **Berechnung**:
   - Der konvertierte `REAL`-Wert wird an den Eingang `in` des `FT_DERIV` Bausteins übergeben.
   - Der `FT_DERIV` Baustein berechnet die Änderung des Eingangssignals pro Zeiteinheit. Da der Eingang ein akkumulierter Zählerstand (Impulse) ist, entspricht die Ableitung der aktuellen Impulsfrequenz (Impulse pro Sekunde/Minute, abhängig von der Zeitbasis).

Lernziele:

- Einbindung von Hardware-Eingängen (LogiBUS).
- Umgang mit Datentyp-Konvertierungen in IEC 61499 / IEC 61131.
- Anwendung mathematischer Funktionen aus der OSCAT-Bibliothek zur Signalverarbeitung.

Zusammenfassung

Die Übung **Uebung_151** zeigt den Aufbau eines Impulszählers mit nachgeschalteter Frequenzberechnung. Durch die Kombination aus Hardware-Treiber, Konvertierungslogik und dem Differenzierer (`FT_DERIV`) wird aus einfachen Zählimpulsen eine nutzbare Prozessgröße (z.B. Drehzahl oder Durchfluss) generiert.

2.2.1.341 Uebung_152: Impulszähler & PI Regler

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_152`. Hier wird eine geschlossene Regelschleife (Closed Loop) implementiert.

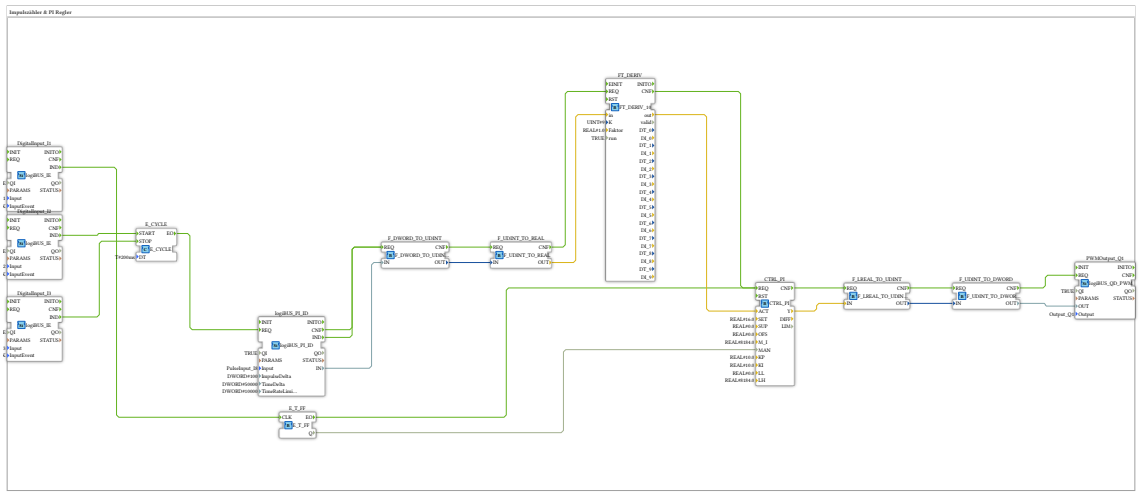
Ziel der Übung

Implementierung eines PI-Reglers zur Konstanthaltung einer physikalischen Größe.

Beschreibung und Komponenten

Die Subapplikation `Uebung_152.SUB` verbindet Sensorik, Regelung und Aktorik.

Regelkreis-Komponenten



- **Sensor (Ist-Wert)**: Impulszähler `logiBUS_PI_ID` + Ableitung `FT_DERIV` (berechnet z.B. die aktuelle Drehzahl).
- **Regler**: `CTRL_PI` (OSCAT). Er vergleicht den Sollwert (`SET = 16.0`) mit dem Ist-Wert.
- **Aktor (Stellgröße)**: `logiBUS_QD_PWM`. Ein pulsweitenmodulierter Ausgang, der z.B. einen Motor oder ein Ventil ansteuert.
- **Bedienung**: Taster `I2` (Start) und `I3` (Stop) steuern den Zyklus. Taster `I1` schaltet zwischen Hand- und Automatikbetrieb um (`MAN` Eingang am Regler).

Funktionsweise

Der Regler versucht ständig, die Stellgröße am PWM-Ausgang so anzupassen, dass die gemessene Impulsrate dem Sollwert entspricht.

- Wird das System belastet (Drehzahl sinkt), erhöht der Regler das PWM-Verhältnis.
- Wird es zu schnell, drosselt er.

Anwendungsbeispiel

Tempomat oder **Konstanthaltung der Ausbringmenge**: Egal ob der Traktor bergauf oder bergab fährt, die Drehzahl der Säwelle soll exakt gleich bleiben.

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

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

2.2.1.342 Uebung_153: Impulszähler & PID Regler

NotebookLM

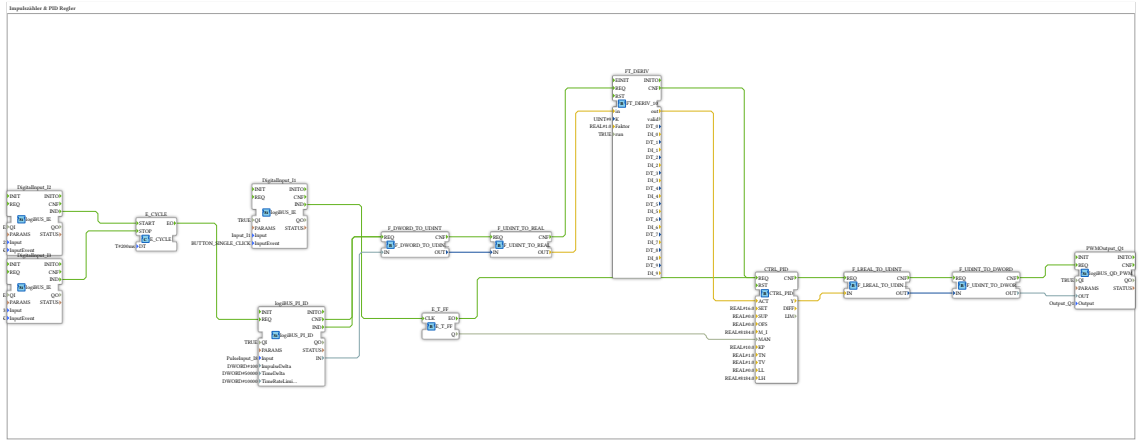
Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_153`.

Ziel der Übung

Präzisere Regelung durch einen PID-Algorithmus.

Beschreibung

Strukturell identisch zu `Uebung_152`, jedoch wird der Baustein `CTRL_PID` verwendet. Zusätzlich zum P- und I-Anteil verfügt dieser über einen D-Anteil (`TV` Parameter), der auf die Änderungsgeschwindigkeit der Regelabweichung reagiert. Dies ermöglicht ein schnelleres Einregeln bei plötzlichen Störungen, erfordert aber eine sorgfältigere Parametrierung.



2.2.1.343 Uebung_160: Motor Links/Rechtslauf

NotebookLM

Dieser Artikel beschreibt die logiBUS®-Übung `Uebung_160`. Hier wird die einfache logische Verknüpfung zur Steuerung eines umschaltbaren Antriebs gezeigt.

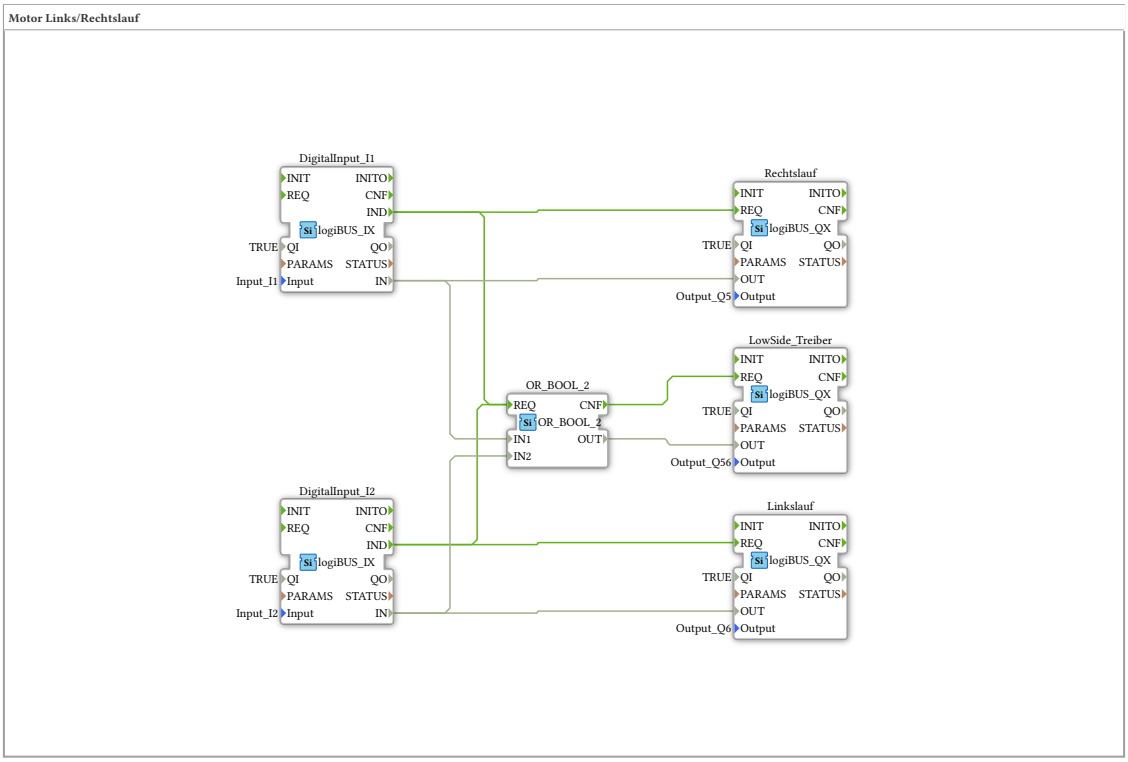
Ziel der Übung

Realisierung einer Steuerung für Linkslauf, Rechtslauf und ein Summensignal (Motor Aktiv).

Beschreibung und Komponenten

In `Uebung_160.SUB` werden zwei Taster auf drei Ausgänge gemappt.

Funktionsbausteine (FBs)



- **`I1`**: Taster für Linkslauf.
- **`I2`**: Taster für Rechtslauf.
- **`OR_2_BOOL`**: Logische ODER-Verknüpfung.
- **`Q5`**: Ausgang Linkslauf.
- **`Q6`**: Ausgang Rechtslauf.
- **`Q56`**: Ausgang „Motor läuft“ (Summe).

Funktionsweise

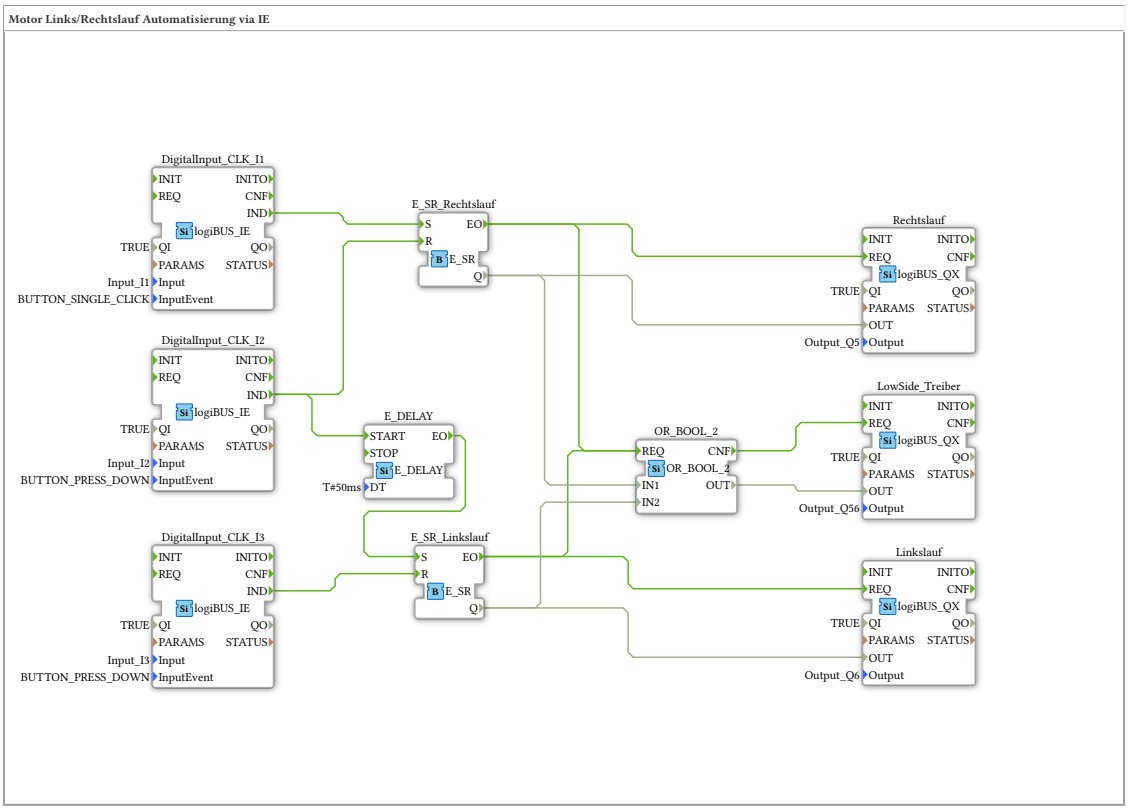
- Drückt der Nutzer **I1**, wird der Ausgang `Q5` aktiv.
- Drückt der Nutzer **I2**, wird der Ausgang `Q6` aktiv.
- Über das ODER-Gatter wird der Ausgang `Q56` immer dann aktiv, wenn **entweder I1 ODER I2** (oder beide) gedrückt werden.

Diese Schaltung demonstriert die Kombination von direkter Signalweiterleitung und logischer Vorverarbeitung für Anzeige-Zwecke.

Anwendungsbeispiel

Zinkenrotor oder Förderband: Die Ausgänge `Q5` und `Q6` steuern die jeweiligen Schütze für die Drehrichtung. Der Ausgang `Q56` steuert eine zentrale Warnleuchte oder ein Entlastungsventil, das immer offen sein muss, wenn der Motor sich in irgendeine Richtung bewegt.

2.2.1.344 Uebung_160b: Motor Links/Rechtslauf Automatisierung via IE



• — ○ • — ○ • — ○ •

Einleitung

Diese Übung implementiert eine Steuerung für einen Motor mit Links- und Rechtslauf (Reversierbetrieb) unter Verwendung von logiBUS-Bausteinen für die Ein- und Ausgabe über Industrial Ethernet (IE). Die Schaltung verfügt über eine direkte Umschaltfunktion mit einer Sicherheitsverzögerung sowie eine Sammelanzeige für den Betriebszustand.

Verwendete Funktionsbausteine (FBs)

In dieser Sub-Application werden verschiedene Standard-Bibliotheksbausteine sowie Hardware-Treiberbausteine verwendet, um die Logik abzubilden.

Sub-Bausteine: Eingänge (Inputs)

Hier werden die Taster für die Steuerung eingelesen.

- **Typ**: `logiBUS::io::DI::logiBUS_IE`
- **Verwendete interne FBs**:
  - **DigitalInput_CLK_I1**: `logiBUS_IE`
    - Parameter: `Input` = `Input_I1`, `InputEvent` = `BUTTON_SINGLE_CLICK`
    - Beschreibung: Startet den ersten Ausgang (Q5). Reagiert auf einfachen Klick.
  - **DigitalInput_CLK_I2**: `logiBUS_IE`
    - Parameter: `Input` = `Input_I2`, `InputEvent` = `BUTTON_PRESS_DOWN`
    - Beschreibung: Stoppt Ausgang Q5 und initiiert den Start von Ausgang Q6. Reagiert auf das Herunterdrücken.
  - **DigitalInput_CLK_I3**: `logiBUS_IE`
    - Parameter: `Input` = `Input_I3`, `InputEvent` = `BUTTON_PRESS_DOWN`
    - Beschreibung: Stoppt den zweiten Ausgang (Q6). Reagiert auf das Herunterdrücken.

Sub-Bausteine: Ausgänge (Outputs)

Diese Bausteine steuern die physischen Ausgänge an.

- **Typ**: `logiBUS::io::DQ::logiBUS_QX`

- **Verwendete interne FBs**:
  - **DigitalOutput_Q5**: `logiBUS_QX`
    - Parameter: `Output` = `Output_Q5`
    - Beschreibung: Steuert den Motor für Richtung A (z.B. Linkslauf).
  - **DigitalOutput_Q6**: `logiBUS_QX`
    - Parameter: `Output` = `Output_Q6`
    - Beschreibung: Steuert den Motor für Richtung B (z.B. Rechtslauf).
  - **DigitalOutput_Q56**: `logiBUS_QX`
    - Parameter: `Output` = `Output_Q56`
    - Beschreibung: Signalleuchte/Statusanzeige, aktiv wenn Q5 oder Q6 aktiv ist.

Sub-Bausteine: Logik (Logic Control)

Die Kernlogik der Steuerung.

- **Typ**: `iec61499::events::E_SR` (Set/Reset Flip-Flop)
- **Verwendete interne FBs**:
  - **E_SR_A**: `E_SR`
    - Funktionsweise: Speichert den Zustand für Ausgang Q5. Wird durch I1 gesetzt und durch I2 zurückgesetzt.
  - **E_SR_B**: `E_SR`
    - Funktionsweise: Speichert den Zustand für Ausgang Q6. Wird verzögert durch I2 gesetzt und durch I3 zurückgesetzt.
- **Typ**: `iec61499::events::E_DELAY` (Verzögerung)
- **Verwendete interne FBs**:
  - **E_DELAY**: `E_DELAY`
    - Parameter: `DT` = `T#50ms`
    - Funktionsweise: Verzögert das Setzen von `E_SR_B` um 50 Millisekunden, nachdem I2 gedrückt wurde. Dies dient vermutlich als kurze Totzeit bei der Umschaltung der Drehrichtung.
- **Typ**: `iec61131::bitwiseOperators::OR_2_BOOL` (Logisches ODER)
- **Verwendete interne FBs**:
  - **OR_2_BOOL**: `OR_2_BOOL`
    - Funktionsweise: Verknüpft die Zustände von Q5 und Q6. Wenn einer der beiden Motorausgänge aktiv ist, wird der Ausgang Q56 aktiviert.

Programmablauf und Verbindungen

Das Netzwerk realisiert eine Motorsteuerung mit folgenden Eigenschaften:

1. **Start Richtung A (Q5):**
   - Wird `Input_I1` (Klick) betätigt, sendet `DigitalInput_CLK_I1` ein Event an den Setz-Eingang (S) von `E_SR_A`.
   - `E_SR_A` setzt seinen Ausgang Q auf TRUE, wodurch `DigitalOutput_Q5` aktiviert wird.
2. **Umschaltung / Stopp A & Start B (Q6):**
   - Wird `Input_I2` (Drücken) betätigt, geschehen zwei Dinge gleichzeitig:
     - Ein Event geht an den Rücksetz-Eingang (R) von `E_SR_A`. Damit wird `DigitalOutput_Q5` sofort ausgeschaltet.
     - Ein Event startet den Timer `E_DELAY`.
   - Nach Ablauf von 50ms (`DT=T#50ms`) sendet `E_DELAY` ein Event an den Setz-Eingang (S) von `E_SR_B`.
   - `E_SR_B` setzt seinen Ausgang Q auf TRUE, wodurch `DigitalOutput_Q6` aktiviert wird.
   - *Hinweis:* I2 fungiert hier als Umschalter von A nach B mit einer kleinen Totzeit.
3. **Stopp Richtung B (Q6):**
   - Wird `Input_I3` (Drücken) betätigt, sendet `DigitalInput_CLK_I3` ein Event an den Rücksetz-Eingang (R) von `E_SR_B`.
   - `DigitalOutput_Q6` wird ausgeschaltet.
4. **Betriebsanzeige (Q56):**

- Die Datenausgänge (Q) von `E_SR_A` und `E_SR_B` sind mit den Eingängen des `OR_2_BOOL` Bausteins verbunden.
- Sobald einer der beiden SR-Speicher aktiv ist (Motor läuft links oder rechts), schaltet `OR_2_BOOL` den `DigitalOutput_Q56` ein.

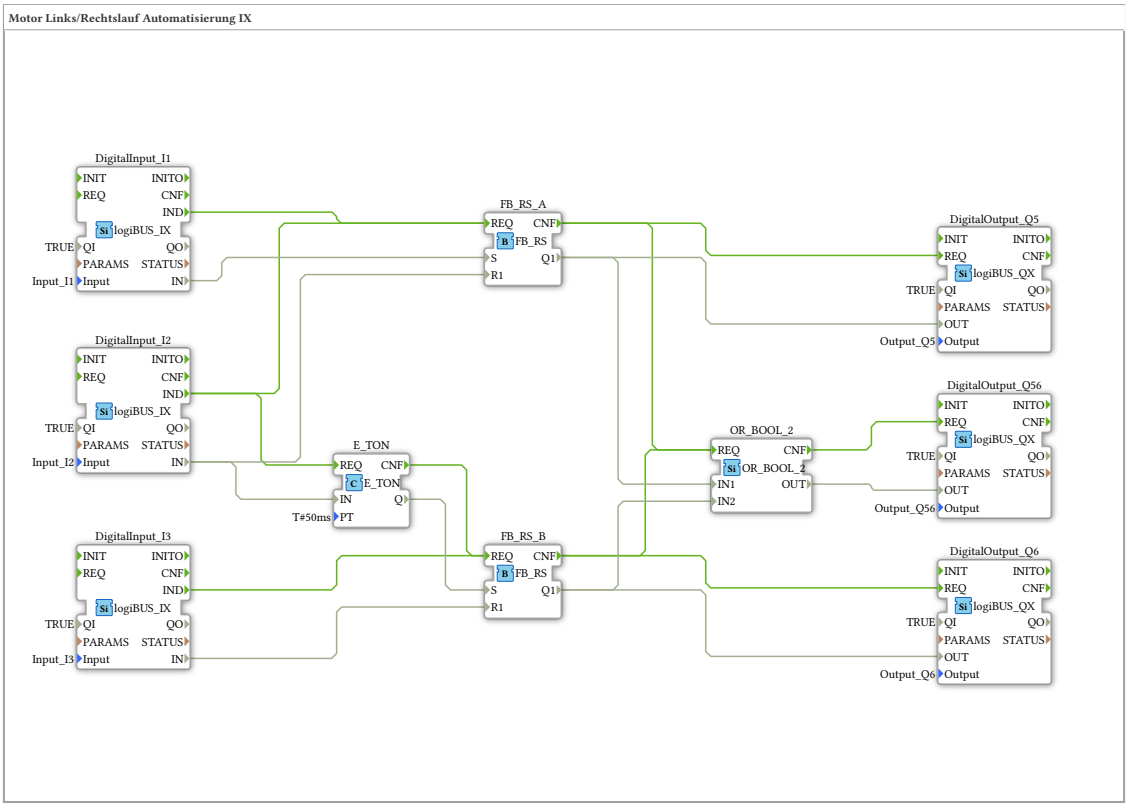
Lernziele:

- Verwendung von bistabilen Kippgliedern (SR-Latch) zur Zustandsspeicherung.
- Implementierung einer Umschaltlogik mit Zeitverzögerung (E_DELAY) zur Vermeidung von abrupten Lastwechseln oder Kurzschlüssen.
- Verarbeitung verschiedener Taster-Events (Single Click vs. Press Down).
- Logische Verknüpfung von Zuständen (OR) zur Ansteuerung einer Sammelanzeige.

Zusammenfassung

Die Übung `Uebung_160b` zeigt eine praxisnahe Implementierung einer Wendeschützschaltunglogik in IEC 61499. Durch die Kombination von SR-Latches und einem Delay-Timer wird sichergestellt, dass beim Umschalten von Links- auf Rechtslauf (ausgelöst durch Taster I2) der erste Ausgang abschaltet, bevor der zweite Ausgang nach 50ms zuschaltet. Taster I1 dient als Start für die erste Richtung, Taster I3 als Stopp für die zweite Richtung. Ausgang Q56 signalisiert, ob der Motor aktuell in Betrieb ist.

2.2.1.345 Uebung_160b2: Motor Links/Rechtslauf Automatisierung IX



(Falls ein Bild der Netzwerksicht vorhanden ist, bitte hier einfügen)

• – ○ • – ○ • – ○ •

Einleitung

Die Übung **Uebung_160b2** implementiert eine Steuerung für einen Motor-Rechts/Linkslauf (Automatisierung IX). Der Fokus dieser Schaltung liegt auf der Umschaltung zwischen zwei Zuständen (Ausgänge Q5 und Q6) mit einer integrierten Totzeit (Verzögerung) beim Wechsel von Q5 auf Q6, gesteuert durch digitale Eingänge. Zusätzlich wird ein Sammelstatus ausgegeben.

Verwendete Funktionsbausteine (FBs)

In dieser Sub-Application werden Standard-Bibliotheksbausteine verwendet, um die logischen Verknüpfungen zu realisieren.

- **logiBUS_IX (DigitalInput_I1, _I2, _I3)**:
  - Stellt die digitalen Eingänge der Hardware (I1, I2, I3) im Netzwerk bereit.
- **logiBUS_QX (DigitalOutput_Q5, _Q6, _Q56)**:
  - Steuert die digitalen Ausgänge der Hardware an.
- **iec61131::bistableElements::FB_RS (FB_RS_A, FB_RS_B)**:
  - Rücksetzdominante bistabile Kippglieder (RS-Flip-Flops). Sie speichern den Zustand „Ein“ oder „Aus“ für die Motoren.
- **iec61131::bitwiseOperators::OR_2_BOOL (OR_2_BOOL)**:
  - Eine logische ODER-Verknüpfung mit zwei Eingängen.
- **iec61499::events::timers::E_TON (E_TON)**:
  - Ein Einschaltverzögerungs-Timer (On-Delay Timer).

Sub-Bausteine

In dieser Übung werden keine benutzerdefinierten Sub-Bausteine verwendet, sondern direkte Instanzen von Standard-FBs verschaltet.

Programmablauf und Verbindungen

Die Schaltung realisiert eine verriegelte Steuerung zweier Ausgänge (z.B. Motor Links/Rechts) mit folgenden Eigenschaften:

1. **Ansteuerung Ausgang Q5 (Erster Pfad):**
   - Der Ausgang **Q5** wird über den Baustein **FB_RS_A** gesteuert.
   - Betätigen von Eingang **I1** setzt (Set) den Baustein und aktiviert Q5.
   - Betätigen von Eingang **I2** setzt den Baustein zurück (Reset) und deaktiviert Q5 sofort.
2. **Umschaltung und Ansteuerung Ausgang Q6 (Zweiter Pfad):**
   - Der Eingang **I2** hat eine Doppelfunktion: Er stoppt Q5 und startet den Vorgang für Q6.
   - Das Signal von **I2** startet den Timer **E_TON**. Dieser ist auf **50ms** (`PT=50ms`) eingestellt.
   - Nach Ablauf der 50ms setzt der Timer-Ausgang (`Q`) den Baustein **FB_RS_B**.
   - Dadurch wird Ausgang **Q6** aktiv.
   - Dies erzeugt eine kurze Totzeit zwischen dem Abschalten von Q5 und dem Einschalten von Q6, was bei Motorsteuerungen wichtig ist, um Kurzschlüsse oder mechanische Belastungen beim direkten Umschalten zu vermeiden.
   - Betätigen von Eingang **I3** setzt **FB_RS_B** zurück und deaktiviert Q6.
3. **Sammelanzeige Q56:**
   - Der Baustein **OR_2_BOOL** überwacht die Ausgänge von FB_RS_A (Q5) und FB_RS_B (Q6).
   - Der Ausgang **Q56** ist aktiv, sobald entweder Q5 **ODER** Q6 aktiv ist. Dies dient als „Betriebsanzeige“.

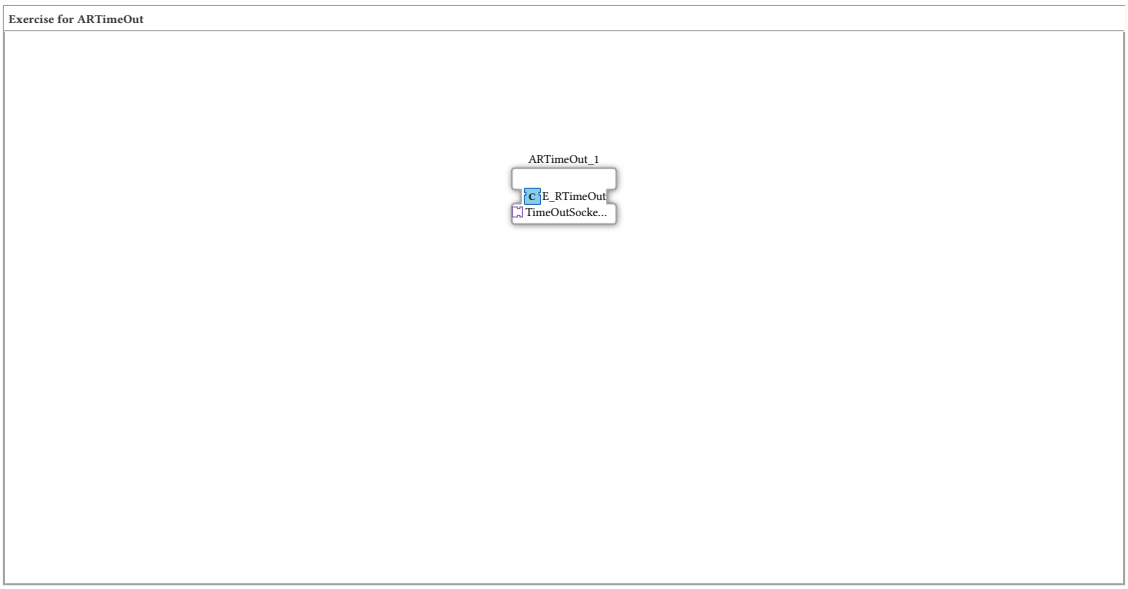
Zusammenfassende Logik:

- **I1** startet Q5.
- **I2** stoppt Q5 und startet (verzögert um 50ms) Q6.
- **I3** stoppt Q6.

Zusammenfassung

Die Übung `Uebung_160b2` demonstriert eine fortgeschrittene Motorsteuerungsschaltung. Sie zeigt, wie man mithilfe von RS-Flip-Flops Zustände speichert und durch den Einsatz eines Timers (`E_TON`) eine automatische Umschaltverzögerung realisiert. Dies ist ein typisches Szenario in der Antriebstechnik, um beim Richtungswechsel die Hardware zu schützen. Die Ausgänge Q5 und Q6 repräsentieren die beiden Laufrichtungen, während Q56 den allgemeinen Betriebszustand signalisiert.

2.2.1.346 Uebung_170: Exercise for ARTimeOut



• – ◦ • – ◦ • – ◦ •

Einleitung

Die `Uebung_170` ist eine Sub-Applikation, die sich mit der Handhabung von Zeitüberschreitungen (Timeouts) befasst. Sie dient als Übungsumgebung für den Funktionsbaustein `E_RTimeOut` (Event Resettable TimeOut), um dessen Verhalten innerhalb eines IEC 61499 Netzwerks zu testen oder zu demonstrieren.

Verwendete Funktionsbausteine (FBs)

In dieser Übung wird eine Instanz eines Ereignis-Bausteins aus der Standardbibliothek verwendet.

Enthaltene Bausteine

- **ARTimeOut_1**
  - **Typ**: `iec61499::events::E_RTimeOut`
  - **Beschreibung**: Dieser Baustein stellt eine Timeout-Funktionalität für Ereignisse bereit. Er wird typischerweise verwendet, um zu überwachen, ob ein Ereignis innerhalb einer bestimmten Zeitspanne eintritt, und kann zurückgesetzt werden.
  - **Konfiguration**:
    - Position im Raster: x=−900, y=300.
    - Es sind im XML keine initialen Parameterwerte für diese Instanz definiert.

Programmablauf und Verbindungen

Aktuell besteht das Netzwerk dieser Übung lediglich aus der Instanz `ARTimeOut_1`.

- **Verbindungen**: Es sind in der vorliegenden Konfiguration keine Verbindungen (weder Daten noch Ereignisse) definiert. Der Baustein steht isoliert im Netzwerk.
- **Funktionsweise der Übung**:
  - Da keine Verbindungen vorhanden sind, ist diese Übung vermutlich dazu gedacht, den Baustein `E_RTimeOut` manuell im Debug-Modus zu testen oder als Ausgangspunkt für eine komplexere Schaltung zu dienen, bei der Verbindungen noch hinzugefügt werden müssen.
  - Der Benutzer kann die Eingangsereignisse des Bausteins manuell triggern, um zu beobachten, wann und wie das Timeout-Ereignis ausgelöst wird.

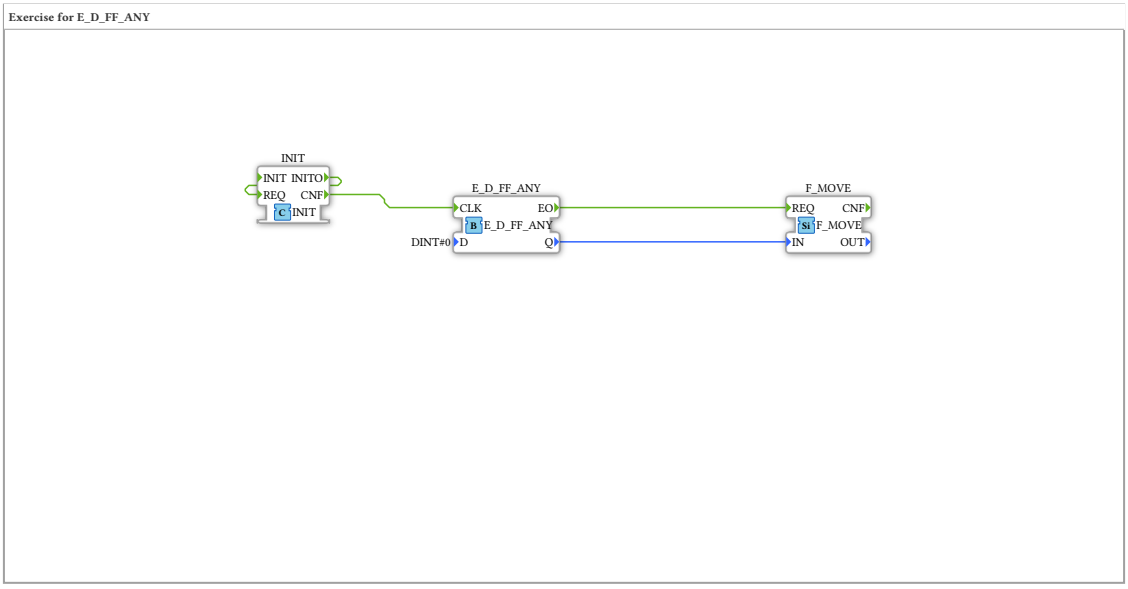
Zusammenfassung

Die `Uebung_170` stellt eine minimale Konfiguration bereit, die ausschließlich den Funktionsbaustein `E_RTimeOut` enthält. Sie dient als grundlegende Vorlage oder Testumgebung, um die Logik von rücksetzbaren Timeouts in der 4diac-IDE zu erlernen oder zu überprüfen.

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

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

2.2.1.347 Uebung_171: Exercise for E_D_FF_ANY



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung (`Uebung_171`) ist als Training für den Umgang mit dem Funktionsbaustein **E_D_FF_ANY** konzipiert. Sie demonstriert das Zusammenspiel zwischen IEC 61131-Funktionen zur Datenmanipulation und IEC 61499-Funktionsbausteinen zur ereignisgesteuerten Datenübertragung.

Verwendete Funktionsbausteine (FBs)

Innerhalb dieser SubApp werden die folgenden Funktionsbausteine verwendet, um die Logik abzubilden:

Sub-Bausteine: Enthaltene Komponenten

In dieser Übung werden spezifisch folgende Bausteine instanziiert:

- **E_D_FF_ANY**
  - **Typ**: `iec61499::events::E_D_FF_ANY`
  - **Beschreibung**: Ein ereignisgesteuerter Baustein, der Daten von einem Eingang zu einem Ausgang bewegt, sobald ein Ereignis ausgelöst wird.
  - **Verwendung in der Übung**: Dient als Empfänger des Datenwerts.
- **F_MOVE**
  - **Typ**: `iec61131::selection::F_MOVE`
  - **Parameter**: `DataType = INT`
  - **Beschreibung**: Eine Standard-IEC 61131 Funktion zur Zuweisung von Werten. In dieser Übung ist der Datentyp explizit auf `INT` (Integer) gesetzt.
  - **Verwendung in der Übung**: Dient als Quelle oder Vorverarbeitung des Datenwerts, der an `E_D_FF_ANY` übergeben wird.

Programmablauf und Verbindungen

Das Netzwerk zeigt eine einfache Verbindung zwischen einer Standard-Funktion und einem Event-Baustein, ist jedoch noch unvollständig (siehe TODO).

Bestehende Datenverbindungen

- **F_MOVE.OUT → E_D_FF_ANY.IN**: Das Ergebnis der Zuweisung/Bewegung aus dem Baustein `F_MOVE` wird direkt an den Dateneingang von `E_D_FF_ANY` geleitet.

Hinweise zur Durchführung

Im Netzwerk befindet sich ein Kommentarbaustein mit dem Inhalt **„TODO“**. Dies deutet darauf hin, dass die Übung vom Anwender vervollständigt werden muss. Wahrscheinlich fehlen:

1. Eingangswerte für `F_MOVE`, um einen Wert zu definieren.
2. Eine Ereignis-Verbindung (Event Connection), um den `E_D_FF_ANY` Baustein zu triggern (Eingang `EI`), damit dieser den Datenwert übernimmt und weitergibt.

Lernziele:

- Verständnis des Unterschieds zwischen reinen Datenfunktionen (`F_MOVE`) und ereignisgesteuerten Bausteinen (`E_D_FF_ANY`).
- Korrekte Verdrahtung von Datentypen (hier `INT`).

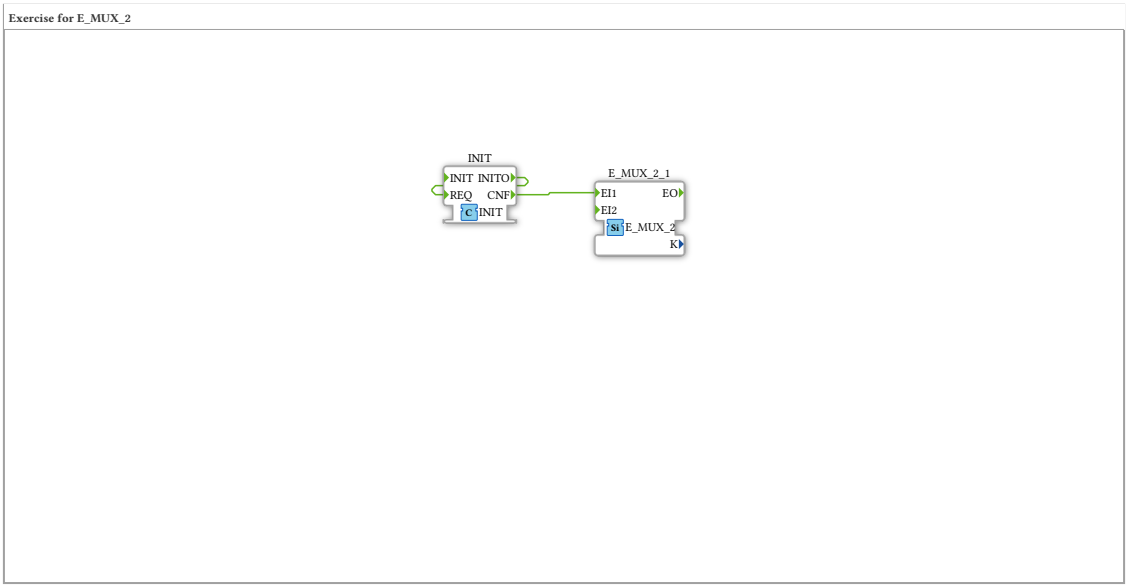
Zusammenfassung

Die `Uebung_171` stellt eine grundlegende Aufgabe dar, um die Datenübergabe in 4diac zu üben. Der Fokus liegt auf der korrekten Nutzung des `E_D_FF_ANY` Bausteins in Kombination mit vorangestellter IEC 61131 Logik (`F_MOVE`). Der Anwender muss die offenen Verbindungen im Sinne des „TODO“-Hinweises ergänzen, um die Funktionalität herzustellen.

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

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

2.2.1.348 Uebung_172: Exercise for E_MUX_2



• – ○ • – ○ • – ○ •

Einleitung

Die Sub-Applikation **Uebung_172** ist eine Übungseinheit, die sich mit dem Funktionsbaustein `E_MUX_2` beschäftigt. Sie dient als Vorlage oder Ausgangspunkt, um die Funktionsweise des Event-Multiplexers (E_MUX) innerhalb der IEC 61499 Norm zu erlernen und zu implementieren.

Verwendete Funktionsbausteine (FBs)

In dieser Übung wird der Standard-Baustein aus der `iec61499::events` Bibliothek verwendet.

Sub-Bausteine: Uebung_172

Diese Sub-Applikation definiert den Rahmen der Übung.

- **Typ**: SubAppType
- **Verwendete interne FBs**:
  - **E_MUX_2_1**: `iec61499::events::E_MUX_2`

- **Beschreibung**: Dies ist eine Instanz des `E_MUX_2` Bausteins (Event Multiplexer für 2 Eingänge).
- **Parameter**: Keine Parameter initial gesetzt.
- **Koordinaten**: x=−3000, y=−1000

Programmablauf und Verbindungen

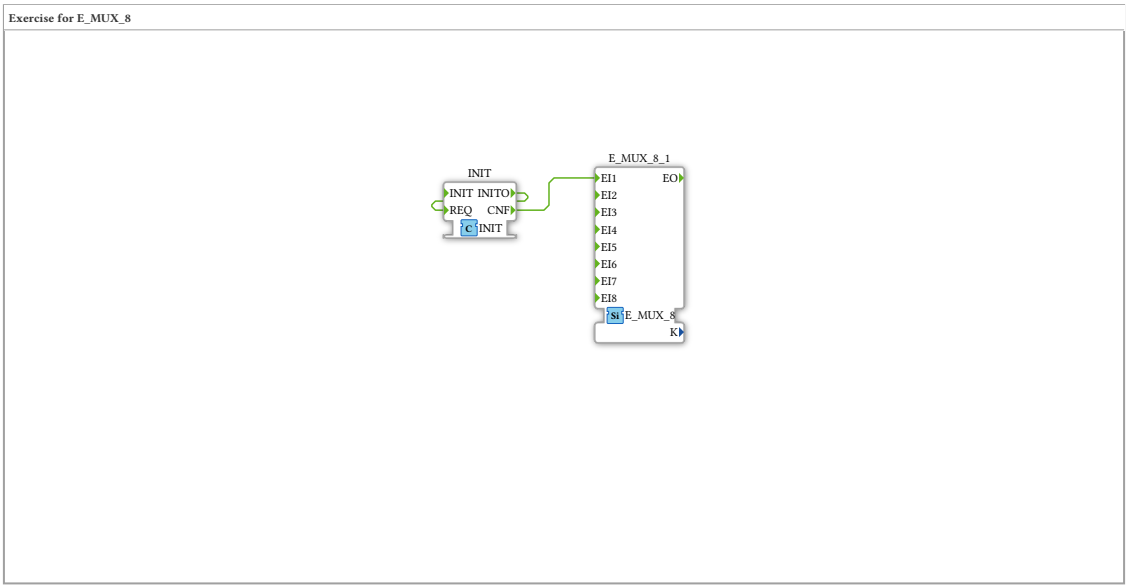
Aktuell stellt die Übung ein Gerüst dar, das vom Anwender vervollständigt werden muss.

- **Status**: Die Sub-Applikation enthält eine Instanz des `E_MUX_2`, jedoch noch keine Verbindungen (Event- oder Datenflüsse).
- **Hinweise**: Im Netzwerk befindet sich ein Kommentar-Feld mit dem Inhalt **„TODO“** (bei x=−1900, y=−400). Dies signalisiert, dass die Logik, Eingänge und Ausgänge noch verdrahtet werden müssen.
- **Ziel der Übung**: Das Ziel ist vermutlich, zwei verschiedene Event-Quellen an den `E_MUX_2` anzuschließen und das resultierende Ausgangs-Event weiterzuverarbeiten, um das Prinzip der Ereignis-Zusammenführung (Multiplexing) zu verstehen.

Zusammenfassung

Die `Uebung_172` ist eine vorbereitete Arbeitsumgebung für die Auseinandersetzung mit dem `E_MUX_2` Funktionsbaustein. Sie enthält die notwendige Instanz des Bausteins und einen Platzhalter-Kommentar, überlässt die konkrete Implementierung der Event-Logik jedoch dem Lernenden.

2.2.1.349 Uebung_173: Exercise for E_MUX_8



• – ◦ • – ◦ • – ◦ •

Einleitung

Die Sub-Applikation **Uebung_173** dient als Übungsumgebung für den Umgang mit dem Funktionsbaustein `E_MUX_8`. Das Ziel dieser Übung ist es, das Konzept des Ereignis-Multiplexings (Zusammenführen mehrerer Ereignispfade) innerhalb der IEC 61499 Norm zu verstehen und anzuwenden.

Der bereitgestellte Arbeitsbereich enthält einen einzelnen Baustein und einen Platzhalter-Kommentar, was darauf hindeutet, dass der Nutzer die Logik vervollständigen muss.

Verwendete Funktionsbausteine (FBs)

In diesem Netzwerk wird primär der folgende Standard-Baustein verwendet:

E_MUX_8_1

- **Typ**: `iec61499::events::E_MUX_8`
- **Beschreibung**: Dies ist ein Ereignis-Multiplexer mit 8 Eingängen.
- **Schnittstellen**:
  - **Ereigniseingänge (EI1 bis EI8)**: Trigger-Eingänge für verschiedene Ereignisquellen.

  - **Ereignisausgang (EO)**: Dieser Ausgang feuert, sobald einer der Eingänge `EI1` bis `EI8` ein Ereignis empfängt.
- **Funktionsweise**: Der Baustein fungiert wie eine ODER-Verknüpfung für Ereignisse. Er leitet jedes eingehende Event an den Ausgang weiter, unabhängig davon, an welchem Eingang es auftritt.

Programmablauf und Verbindungen

Aktuell befindet sich die Übung in einem initialen Zustand („TODO“).

🌐 **Netzwerk-Status**

- **Vorhandene Instanzen**: Eine Instanz des Multiplexers (`E_MUX_8_1`) ist im Netzwerk platziert (Koordinaten: −3000, −1000).
- **Verbindungen**: Es sind **keine** Verbindungen im XML definiert. Der Baustein steht isoliert.
- **Kommentare**: Ein großer Kommentarblock mit dem Inhalt „TODO“ markiert den Bereich, in dem die Implementierung erfolgen soll.

Durchführung der Übung

1. **Ziel**: Der Nutzer soll vermutlich Ereignisquellen (z.B. von anderen Bausteinen oder Eingängen der SubApp) mit den Eingängen des `E_MUX_8` verbinden.
2. **Logik**: Es soll realisiert werden, dass verschiedene Ereignisse auf einen einzigen Ereignispfad (den Ausgang des Mux) zusammengeführt werden.
3. **Voraussetzungen**: Verständnis darüber, wie Events in 4diac verbunden werden und wie die Abarbeitungsreihenfolge (Execution Order) funktioniert.

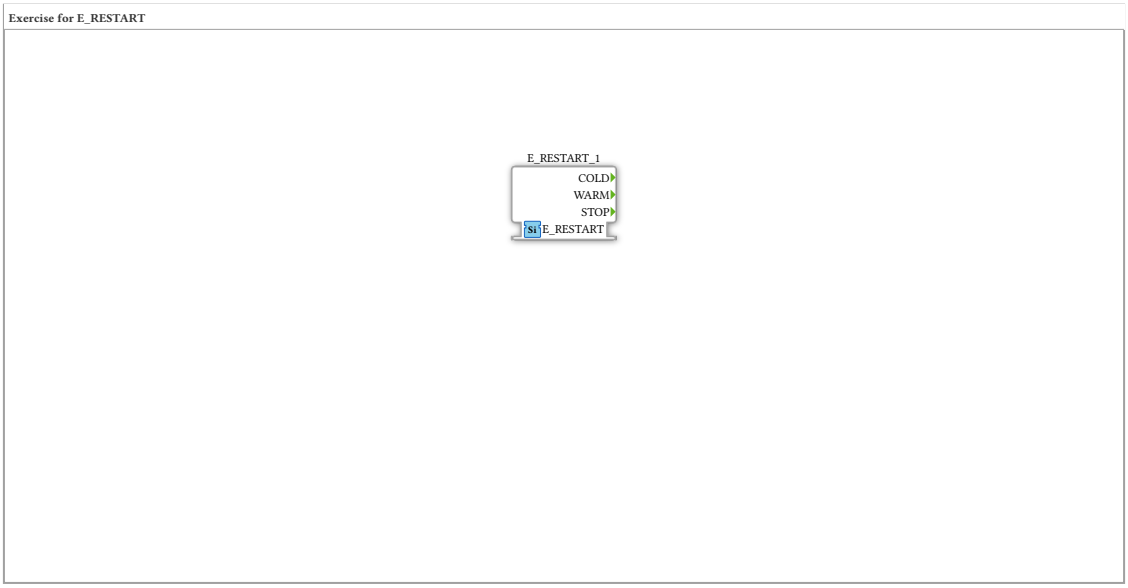
Zusammenfassung

Die `Uebung_173` ist eine grundlegende Vorlage zur Erlernung der Ereignissteuerung mittels `E_MUX_8`. Sie stellt den notwendigen Baustein bereit, überlässt aber die Verschaltung und Integration in eine größere Logik dem Anwender als Teil der Lernaufgabe.

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

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

2.2.1.350 Uebung_174: Exercise for E_RESTART



• – ◦ • – ◦ • – ◦ •

Einleitung

Die **Uebung_174** ist eine Sub-Application, die sich mit dem Initialisierungsverhalten von Steuerungen in IEC 61499 beschäftigt. Spezifisch geht es um den Umgang mit dem `E_RESTART` Funktionsbaustein. Die Übung stellt ein Grundgerüst dar, in dem die Logik für Kalt- und Warmstarts implementiert werden soll.

Verwendete Funktionsbausteine (FBs)

In dieser Übung wird primär ein spezifischer Event-Baustein aus der Standardbibliothek verwendet.

Enthaltene Bausteine

- **E_RESTART_1**
  - **Typ**: `iec61499::events::E_RESTART`
  - **Beschreibung**: Dieser Funktionsbaustein stellt Ereignisse zur Verfügung, die ausgelöst werden, wenn die Ressource, auf der die Applikation läuft, gestartet wird.
    - **Ereignisausgang COLD**: Wird ausgelöst bei einem Kaltstart (erstmaliges Starten oder Reset).
    - **Ereignisausgang WARM**: Wird ausgelöst bei einem Warmstart (Wiederaufnahme des Betriebs, falls unterstützt).
  - **Funktionsweise**: Er dient als Trigger für Initialisierungsroutinen innerhalb der Applikation.

Programmablauf und Verbindungen

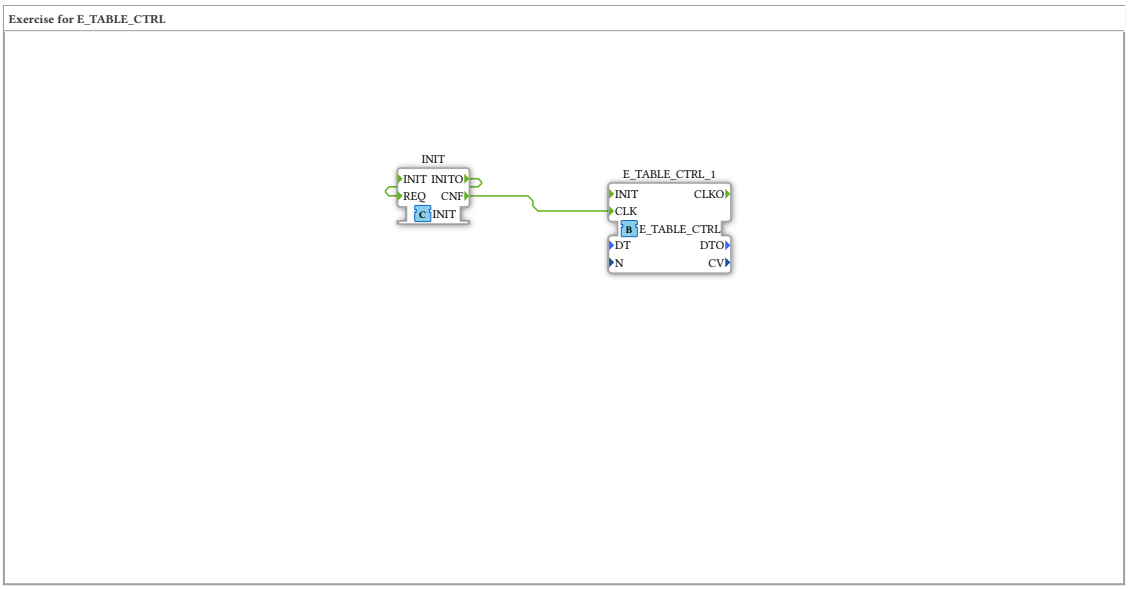
Aktuell stellt diese Übung ein leeres Netzwerk mit einer Aufgabenstellung dar.

- **Aktueller Zustand**:
  - Der Baustein `E_RESTART_1` ist im Netzwerk platziert.
  - Es existieren noch keine Verbindungen zu anderen Bausteinen.
  - Ein Kommentarfeld mit dem Inhalt **„TODO“** weist auf den zu bearbeitenden Bereich hin.
- **Lernziele**:
  - Verständnis des Unterschieds zwischen `COLD` und `WARM` Start-Events.
  - Nutzung des `E_RESTART` Bausteins zur Initialisierung von Variablen oder Zuständen beim Start der Steuerung.
- **Vorgehensweise**:
  1. Die Übung wird als SubApp geöffnet.
  2. An die Ausgänge des `E_RESTART_1` Bausteins sollen Logik-Ketten angeschlossen werden, die definieren, was beim Start der Applikation passieren soll (z.B. Setzen von Standardwerten).

Zusammenfassung

Die `Uebung_174` ist eine Basisübung zur Implementierung von Start-up-Routinen. Sie bietet den `E_RESTART` Baustein an und fordert den Anwender durch einen „TODO“-Kommentar dazu auf, die entsprechende Initialisierungslogik für den Kalt- und Warmstart der Steuerung zu entwickeln.

2.2.1.351 Uebung_175: Exercise for E_TABLE_CTRL



• – ◦ • – ◦ • – ◦ •

Einleitung

Die Übung **Uebung_175** ist eine Vorlage zur Einarbeitung in die Nutzung von Tabellensteuerungen innerhalb der IEC 61499 Architektur. Der Fokus liegt speziell auf dem Funktionsbaustein `E_TABLE_CTRL` (Event Table Control). Die Übung stellt ein Grundgerüst bereit, das durch den Anwender vervollständigt werden muss.

Verwendete Funktionsbausteine (FBs)

In dieser Sub-Applikation wird primär ein Instanz eines Standard-Bibliotheksbausteins verwendet.

Sub-Bausteine: E_TABLE_CTRL_1

- **Typ**: `iec61499::events::E_TABLE_CTRL`
- **Verwendete interne FBs**:
  - Dieser Baustein ist eine Instanz aus der Standardbibliothek (`iec61499`).
  - **Parameter**:
    - Aktuell sind keine Parameter im Netzwerk vordefiniert.
  - **Funktionsweise**:
    - Der `E_TABLE_CTRL` Baustein dient üblicherweise dazu, ereignisgesteuerte Abläufe basierend auf einer Zustandstabelle oder einer definierten Sequenz zu steuern. Er schaltet Ausgänge basierend auf Eingangszuständen und einer hinterlegten Logik (oft State-Machine ähnlich).

Programmablauf und Verbindungen

Das Netzwerk dieser Übung ist als **Aufgabe (TODO)** konzipiert.

- **Status des Netzwerks**:
  - Der Baustein `E_TABLE_CTRL_1` wurde im Netzwerk bei den Koordinaten `x=-3000, y=-1000` platziert.
  - Es existieren aktuell **keine Verbindungen** (weder Daten noch Ereignisse) zwischen Bausteinen oder Schnittstellen.
  - Ein großer Kommentarblock mit dem Inhalt **„TODO“** (bei `x=-3100, y=-100`) weist darauf hin, dass die eigentliche Implementierung der Steuerungslogik noch erfolgen muss.
- **Lernziele**:
  - Verständnis der Schnittstellen des `E_TABLE_CTRL`.
  - Verschaltung von Ereignissen und Daten zur Realisierung einer Sequenzsteuerung.
- **Vorgehensweise**:
  1. Analysieren Sie die benötigten Ein- und Ausgänge des `E_TABLE_CTRL`.

2. Verbinden Sie die notwendigen Ereignis- und Datenleitungen gemäß der Aufgabenstellung (die hier implizit durch das „TODO“ gegeben ist).
3. Konfigurieren Sie die Parameter des Bausteins, falls notwendig.

Zusammenfassung

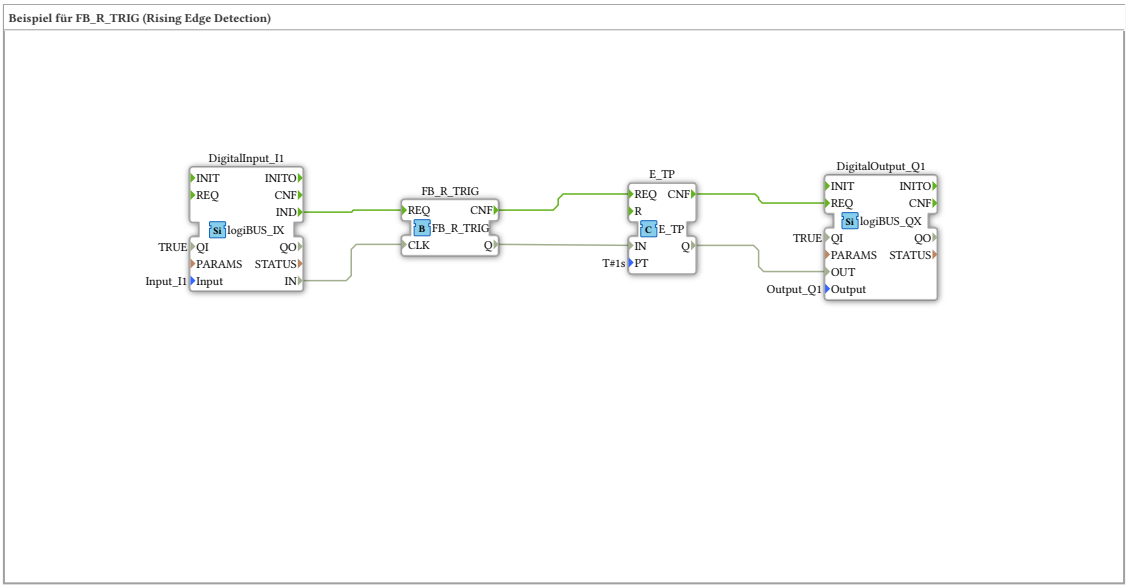
Die `Uebung_175` ist eine leere Übungsumgebung („Skeleton“), die lediglich den Baustein `E_TABLE_CTRL` bereitstellt. Ziel der Übung ist es, die Funktionalität dieses Bausteins durch eigenständiges Erstellen der Verbindungen und Logik zu erlernen.

2.2.1.352 Uebung_176

Hinweis

Diese Übung existiert in den aktuellen Trainingsquellen nicht mehr (entfernt, zusammengelegt oder umbenannt). Diese Seite bleibt nur als Platzhalter für bestehende Verweise erhalten. Die aktuellen Übungen stehen in der Übungsübersicht.

2.2.1.353 Uebung_177: Beispiel für FB_R_TRIG (Rising Edge Detection)



• – ○ • – ○ • – ○ •

Einleitung

Die **Uebung_177** demonstriert die Verwendung des `FB_R_TRIG` Funktionsbausteins zur Erkennung einer steigenden Flanke (Rising Edge Detection). Ziel der Übung ist es, ein Eingangssignal so zu verarbeiten, dass nur der Moment des Einschaltens (Wechsel von FALSE auf TRUE) eine Aktion auslöst. Dieses kurzzeitige Signal wird anschließend genutzt, um einen Timer zu starten, der einen Ausgang für eine definierte Zeit aktiviert.

Verwendete Funktionsbausteine (FBs)

In dieser Übung werden folgende Funktionsbausteine innerhalb des Netzwerks verwendet:

- **DigitalInput_I1** (`logiBUS::io::DI::logiBUS_IX`)
  - Dient als Schnittstelle zum physischen Eingang `Input_I1`. Er liefert das Eingangssignal für die Flankenerkennung.
- **FB_R_TRIG** (`iec61131::edgeDetection::FB_R_TRIG`)
  - Dies ist der Kernbaustein der Übung. Er überwacht den Eingang `CLK`. Wenn `CLK` von FALSE auf TRUE wechselt (steigende Flanke), wird der Ausgang `Q` für genau einen Zyklus auf TRUE gesetzt.
- **E_TP** (`iec61499::events::timers::E_TP`)
  - Ein Impuls-Timer (Pulse Timer).
  - **Parameter**: `PT` ist auf `T#1s` (1 Sekunde) eingestellt.
  - Erzeugt einen Impuls von 1 Sekunde Länge, sobald der Eingang `IN` aktiviert wird.
- **DigitalOutput_Q1** (`logiBUS::io::DQ::logiBUS_QX`)

– Dient als Schnittstelle zum physischen Ausgang `Output_Q1`. Er schaltet den Ausgang basierend auf dem Signal des Timers.

Programmablauf und Verbindungen

Der Ablauf der Steuerung gestaltet sich wie folgt:

1. **Signalerfassung**: Der Baustein `DigitalInput_I1` liest den Zustand des physikalischen Eingangs (z.B. ein Taster). Das Ereignis `IND` und der Datenwert `IN` werden an den Flankenerkennungs-Baustein weitergeleitet.
2. **Flankenerkennung**:
   - Der Baustein `FB_R_TRIG` empfängt das Signal am Eingang `CLK`.
   - Sobald eine Änderung von 0 auf 1 (Taster wird gedrückt) erkannt wird, setzt `FB_R_TRIG` seinen Ausgang `Q` kurzzeitig auf TRUE.
   - Wird der Taster gehalten oder losgelassen (fallende Flanke), bleibt der Ausgang `Q` auf FALSE.
3. **Zeitsteuerung**:
   - Das kurzzeitige Signal von `FB_R_TRIG.Q` triggert den Eingang `IN` des Timers `E_TP`.
   - Der Timer `E_TP` startet daraufhin einen Impuls. Sein Ausgang `Q` wird für die Dauer von 1 Sekunde (`PT=T#1s`) auf TRUE gesetzt, unabhängig davon, ob das Eingangssignal am Taster noch anliegt oder nicht.
4. **Ausgabe**:
   - Der Zustand des Timers (`E_TP.Q`) wird an `DigitalOutput_Q1.OUT` übergeben.
   - Dies bewirkt, dass der physikalische Ausgang `Output_Q1` (z.B. eine Lampe) für genau 1 Sekunde leuchtet, jedes Mal wenn der Eingangstaster neu gedrückt wird.

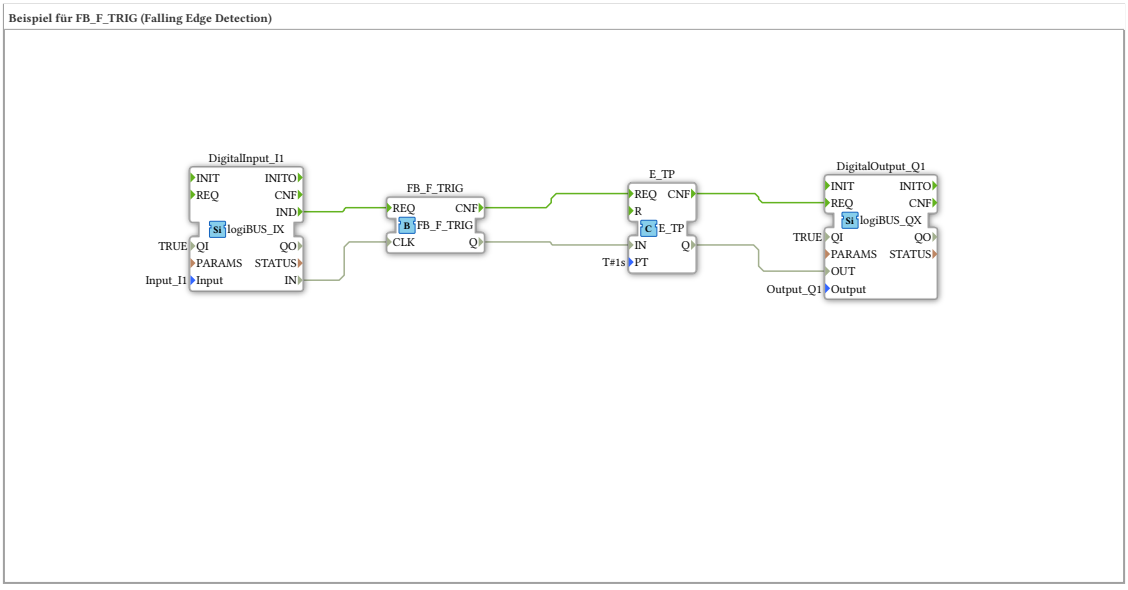
Verbindungsübersicht:

- `DigitalInput_I1.IND` -> `FB_R_TRIG.REQ`
- `DigitalInput_I1.IN` -> `FB_R_TRIG.CLK`
- `FB_R_TRIG.CNF` -> `E_TP.REQ`
- `FB_R_TRIG.Q` -> `E_TP.IN`
- `E_TP.CNF` -> `DigitalOutput_Q1.REQ`
- `E_TP.Q` -> `DigitalOutput_Q1.OUT`

Zusammenfassung

Diese Übung zeigt eine klassische Anwendung in der Automatisierungstechnik: Das Entkoppeln eines statischen Eingangssignals (Schalterzustand) von der Ausgabelogik durch Flankenerkennung. Durch die Kombination von `FB_R_TRIG` und `E_TP` wird sichergestellt, dass der Ausgang `Q1` bei jedem Drücken des Tasters `I1` exakt für eine Sekunde aktiv ist, selbst wenn der Taster länger gedrückt gehalten wird.

2.2.1.354 Uebung_178: Beispiel für FB_F_TRIG (Falling Edge Detection)



• – ◦ • – ◦ • – ◦ •

Einleitung

Die **Uebung_178** beschäftigt sich mit der Erkennung von fallenden Flanken in der Signalverarbeitung. Ziel ist es, ein Ereignis genau dann auszulösen, wenn ein Eingangssignal von `TRUE` (High) auf `FALSE` (Low) wechselt. Zusätzlich wird dieses Ereignis genutzt, um einen zeitlich begrenzten Impuls zu generieren.

Verwendete Funktionsbausteine (FBs)

In dieser Sub-Application werden folgende Funktionsbausteine verwendet:

- **DigitalInput_I1** (`logiBUS::io::DI::logiBUS_IX`):
  - Dient zum Einlesen des digitalen Signals.
  - Konfiguriert auf den Hardware-Eingang `Input_I1`.
- **FB_F_TRIG** (`iec61131::edgeDetection::FB_F_TRIG`):
  - Baustein zur Erkennung einer fallenden Flanke (Falling Edge Trigger).
- **E_TP** (`iec61499::events::timers::E_TP`):
  - Ein Impulsgeber (Pulse Timer).
  - Konfiguriert mit einer Zeitdauer (`PT`) von 1 Sekunde (`T#1s`).
- **DigitalOutput_Q1** (`logiBUS::io::DQ::logiBUS_QX`):
  - Dient zur Ausgabe des digitalen Signals.
  - Konfiguriert auf den Hardware-Ausgang `Output_Q1`.

Programmablauf und Verbindungen

Der Programmablauf dieser Übung gestaltet sich wie folgt:

1. **Signaleingang:** Der Baustein `DigitalInput_I1` liest den Status des Hardware-Eingangs `Input_I1`. Sobald ein Signal anliegt (z.B. ein Taster gedrückt wird) oder sich ändert, wird dies über das Event `IND` und den Datenanschluss `IN` weitergegeben.
2. **Flankenerkennung:** Das Eingangssignal (`IN` von `DigitalInput_I1`) ist mit dem Takteingang (`CLK`) des `FB_F_TRIG` verbunden.
   - Der `FB_F_TRIG` überwacht dieses Signal.
   - Erkennt der Baustein einen Wechsel von **High auf Low** (z.B. Loslassen eines Tasters), schaltet der Ausgang `Q` kurzzeitig auf `TRUE`.
3. **Zeitsteuerung (Impuls):** Das Ausgangssignal `Q` des Flankentriggers ist mit dem Eingang `IN` des Timers `E_TP` verbunden.
   - Sobald die fallende Flanke erkannt wurde, startet der Timer `E_TP`.
   - Der Timer generiert einen Impuls mit der Dauer von **1 Sekunde** (definiert durch `PT = T#1s`).
4. **Signalausgang:** Der Ausgang `Q` des Timers steuert den Eingang `OUT` des `DigitalOutput_Q1`.
   - Dies bewirkt, dass der Hardware-Ausgang `Output_Q1` (z.B. eine Lampe) für genau 1 Sekunde aktiviert wird, nachdem das Eingangssignal abgefallen ist.

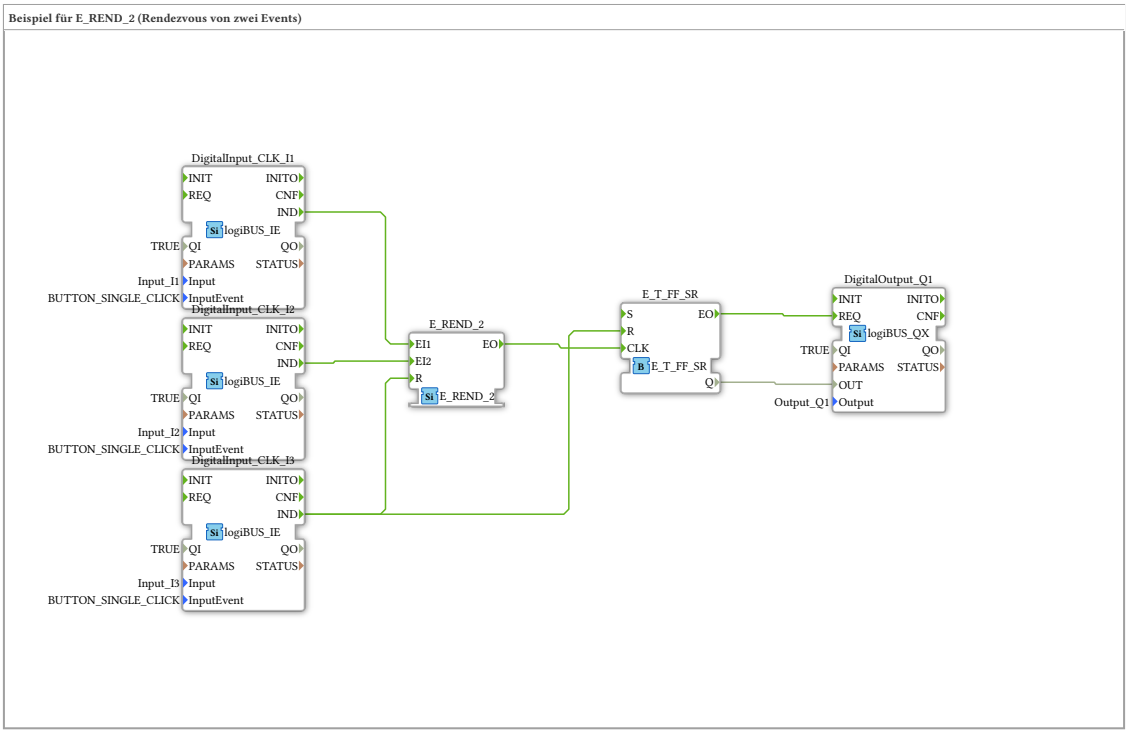
Zusammenfassender Datenfluss: `DigitalInput_I1.IN` -> `FB_F_TRIG.CLK` -> `FB_F_TRIG.Q` -> `E_TP.IN` -> `E_TP.Q` -> `DigitalOutput_Q1.OUT`

Zusammenfassender Ereignisfluss: `DigitalInput_I1.IND` -> `FB_F_TRIG.REQ` -> `FB_F_TRIG.CNF` -> `E_TP.REQ` -> `E_TP.CNF` -> `DigitalOutput_Q1.REQ`

Zusammenfassung

Die Uebung_178 demonstriert die klassische Anwendung einer „Nachlaufsteuerung“ oder Abschaltverzögerung basierend auf einem negativen Signalwechsel. Der Benutzer lernt hierbei die Kombination aus digitaler Signalerfassung, logischer Flankenauswertung mittels `FB_F_TRIG` und zeitgesteuerter Ausgabe mittels `E_TP`. Ein praktisches Beispiel wäre ein Licht, das für eine Sekunde aufleuchtet, sobald ein Taster *losgelassen* wird.

2.2.1.355 Uebung_179: Beispiel für E_REND_2 (Rendezvous von zwei Events)



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung demonstriert die Verwendung des Funktionsbausteins **E_REND_2** (Event Rendezvous). Ziel ist es, das Konzept der Event-Synchronisation zu verstehen. Ein Rendezvous-Baustein sorgt dafür, dass ein ausgehendes Ereignis erst dann ausgelöst wird, wenn an beiden Eingängen ein Ereignis eingetroffen ist. Dies wird oft verwendet, um zwei parallele Prozesse zu synchronisieren, bevor ein nachfolgender Schritt ausgeführt wird.

Verwendete Funktionsbausteine (FBs)

In dieser Sub-Applikation werden folgende Funktionsbausteine aus der Standard- und logiBUS-Bibliothek verwendet:

- **logiBUS::io::DI::logiBUS_IE (3x)**
  - Verwendet als: `DigitalInput_CLK_I1`, `DigitalInput_CLK_I2`, `DigitalInput_CLK_I3`
  - Funktion: Stellt die physikalischen Eingänge (Taster) I1, I2 und I3 bereit. Konfiguriert auf das Event `BUTTON_SINGLE_CLICK`.
- **iec61499::events::E_REND_2**
  - Verwendet als: `E_REND_2`
  - Funktion: Ein Event-Synchronisierungs-Baustein. Er wartet auf Ereignisse an den Eingängen `EI1` und `EI2`. Erst wenn beide Ereignisse eingetroffen sind (unabhängig von der Reihenfolge), wird das Ausgangsereignis `EO` gefeuert. Über den Eingang `R` kann der interne Status zurückgesetzt werden.
- **iec61499::events::E_T_FF_SR**
  - Verwendet als: `E_T_FF_SR`
  - Funktion: Ein Toggle-Flip-Flop (T-Flip-Flop) mit Set- und Reset-Eingängen. Bei jedem Ereignis am Eingang `CLK` wechselt der Zustand des Ausgangs `Q` (von TRUE auf FALSE oder umgekehrt).
- **logiBUS::io::DQ::logiBUS_QX**
  - Verwendet als: `DigitalOutput_Q1`
  - Funktion: Steuert den physikalischen Ausgang Q1 an, um den aktuellen Status anzuzeigen.

Programmablauf und Verbindungen

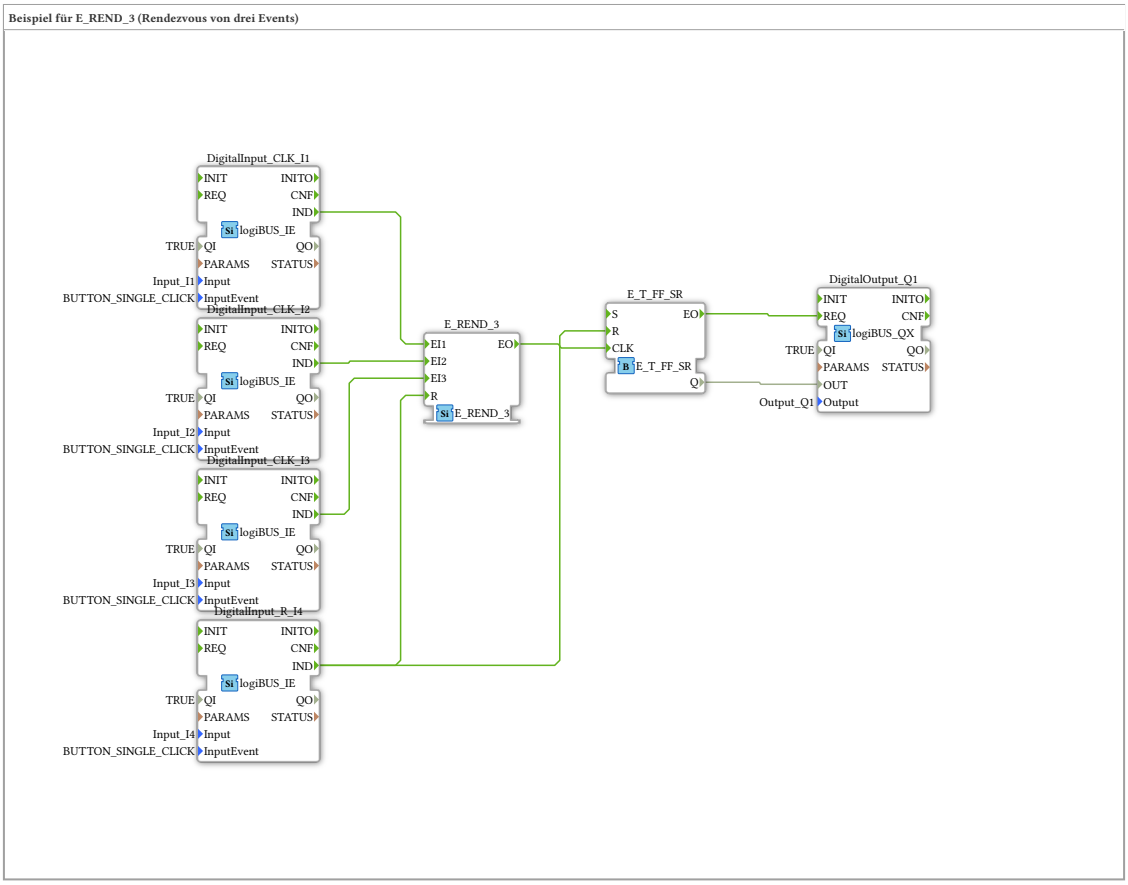
Der Ablauf der Übung gestaltet sich wie folgt:

1. **Eingänge (Rendezvous):**
   - Der Taster **I1** (`DigitalInput_CLK_I1`) ist mit dem ersten Ereigniseingang `EI1` des `E_REND_2` Bausteins verbunden.
   - Der Taster **I2** (`DigitalInput_CLK_I2`) ist mit dem zweiten Ereigniseingang `EI2` des `E_REND_2` Bausteins verbunden.
   - Das Drücken von nur einem dieser beiden Taster bewirkt zunächst keine Änderung am Ausgang. Der Baustein „merkt“ sich das Ereignis.
   - Erst wenn **beide** Taster (I1 und I2) betätigt wurden (das „Rendezvous“ der Events ist komplett), löst der `E_REND_2` das Ausgangsevent `EO` aus.
2. **Verarbeitung (Toggle):**
   - Das Ausgangsevent `EO` des `E_REND_2` ist mit dem Takteingang `CLK` des `E_T_FF_SR` verbunden.
   - Sobald das Rendezvous stattgefunden hat, wird das Flip-Flop getriggert und schaltet den Ausgang Q1 um (Licht an oder aus).
3. **Reset-Funktion:**
   - Der Taster **I3** (`DigitalInput_CLK_I3`) fungiert als zentraler Reset.
   - Er ist mit dem Reset-Eingang `R` des `E_REND_2` verbunden. Dies löscht eventuell bereits gespeicherte Einzel-Events (z.B. wenn I1 gedrückt wurde, aber I2 noch fehlt).
   - Gleichzeitig ist I3 mit dem Reset-Eingang `R` des Flip-Flops `E_T_FF_SR` verbunden, wodurch der Ausgang Q1 sofort ausgeschaltet wird (`FALSE`).
4. **Ausgabe:**
   - Der Datenstatus `Q` des Flip-Flops wird an den `DigitalOutput_Q1` übertragen und steuert die Hardware-LED/Lampe an.

Zusammenfassung

Die Übung `Uebung_179` zeigt anschaulich, wie man zwei unabhängige Ereignisströme synchronisiert. Die Lampe Q1 schaltet ihren Zustand nur um, wenn sowohl Taster I1 als auch Taster I2 betätigt wurden. Mit Taster I3 kann der Prozess jederzeit abgebrochen und der Ausgang zurückgesetzt werden. Dies ist ein grundlegendes Muster für Steuerungen, bei denen zwei Bedingungen (Events) erfüllt sein müssen, um fortzufahren.

2.2.1.356 Uebung_180: Beispiel für E_REND_3 (Rendezvous von drei Events)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert die Synchronisation von Ereignissen mittels eines Rendezvous-Bausteins. Ziel ist es, eine Aktion (das Umschalten eines Ausgangs) erst dann auszuführen, wenn drei separate Eingangsereignisse eingetreten sind. Dies veranschaulicht das Prinzip der Ereignis-Synchronisation in IEC 61499 Steuerungssystemen.

Verwendete Funktionsbausteine (FBs)

In dieser SubApplikation werden verschiedene Funktionsbausteine kombiniert, um die Logik abzubilden. Da es sich bei `Uebung_180` um einen SubAppType handelt, werden hier die darin enthaltenen internen Bausteine und ihre Konfiguration beschrieben.

Interne Bausteine

`DigitalInput_CLK_I1`, `DigitalInput_CLK_I2`, `DigitalInput_CLK_I3`

- **Typ**: `logiBUS::io::DI::logiBUS_IE`
- **Funktion**: Stellen die drei notwendigen Eingangssignale bereit, die synchronisiert werden sollen.
- **Konfiguration**:
  - `Input` = `Input_I1` / `Input_I2` / `Input_I3`
  - `InputEvent` = `BUTTON_SINGLE_CLICK`
  - `QI` = `TRUE`

`DigitalInput_R_I4`

- **Typ**: `logiBUS::io::DI::logiBUS_IE`
- **Funktion**: Dient als zentraler Reset-Eingang für die Schaltung.
- **Konfiguration**:
  - `Input` = `Input_I4`

  - `InputEvent` = `BUTTON_SINGLE_CLICK`
  - `QI` = `TRUE`

`E_REND_3`

- **Typ**: `iec61499::events::E_REND_3`
- **Funktion**: Ein „Rendezvous"-Baustein für drei Ereignisse. Er wartet, bis an allen drei Eingängen (`EI1`, `EI2`, `EI3`) jeweils ein Ereignis eingetreten ist. Erst wenn alle drei Ereignisse registriert wurden (die Reihenfolge ist dabei irrelevant), feuert der Ausgang `EO`.
- **Anschlüsse**:
  - Eingänge `EI1`, `EI2`, `EI3` verbunden mit den digitalen Eingängen I1, I2 und I3.
  - Eingang `R` verbunden mit Reset-Eingang I4.

`E_T_FF_SR`

- **Typ**: `iec61499::events::E_T_FF_SR`
- **Funktion**: Ein Toggle-Flip-Flop (T-FlipFlop) mit Set- und Reset-Eingängen. Bei jedem Ereignis am `CLK`-Eingang wechselt der Status des Ausgangs `Q`.
- **Anschlüsse**:
  - `CLK` verbunden mit dem Ausgang des Rendezvous-Bausteins.
  - `R` (Reset) verbunden mit dem Reset-Eingang I4.

`DigitalOutput_Q1`

- **Typ**: `logiBUS::io::DQ::logiBUS_QX`
- **Funktion**: Steuert den physischen Ausgang basierend auf dem Status des Flip-Flops.
- **Konfiguration**:
  - `Output` = `Output_Q1`
  - `QI` = `TRUE`

Programmablauf und Verbindungen

Die Schaltung realisiert eine logische UND-Verknüpfung auf zeitlicher Ebene (Synchronisation):

1. **Eingabeerfassung**: Die drei Eingangsbausteine `DigitalInput_CLK_I1`, `_I2` und `_I3` senden bei Betätigung (Single Click) ein `IND`-Event.
2. **Rendezvous (Synchronisation)**: Diese drei Events werden an den Baustein `E_REND_3` geleitet.
   - Der Baustein speichert intern, welche Eingänge bereits betätigt wurden.
   - Erst wenn **alle drei** Eingänge (I1, I2 und I3) mindestens einmal ein Signal gesendet haben, wird das Ausgangsevent `EO` des `E_REND_3` ausgelöst.
3. **Verarbeitung (Toggle)**: Das `EO`-Event des Rendezvous-Bausteins triggert den `CLK`-Eingang des `E_T_FF_SR`.
   - Das Flip-Flop wechselt seinen Zustand (von FALSE auf TRUE oder umgekehrt).
   - Der neue Zustand `Q` wird an den Ausgang `DigitalOutput_Q1` übergeben, wodurch die Lampe (Q1) an- oder ausgeht.
4. **Reset**: Der Eingang `DigitalInput_R_I4` ist mit den Reset-Eingängen (`R`) sowohl des `E_REND_3` als auch des `E_T_FF_SR` verbunden.
   - Ein Signal an I4 löscht den internen Speicher des Rendezvous-Bausteins (es müssen erneut alle 3 Taster gedrückt werden).
   - Gleichzeitig wird das Flip-Flop zurückgesetzt, wodurch der Ausgang Q1 sofort auf `FALSE` (Aus) schaltet.

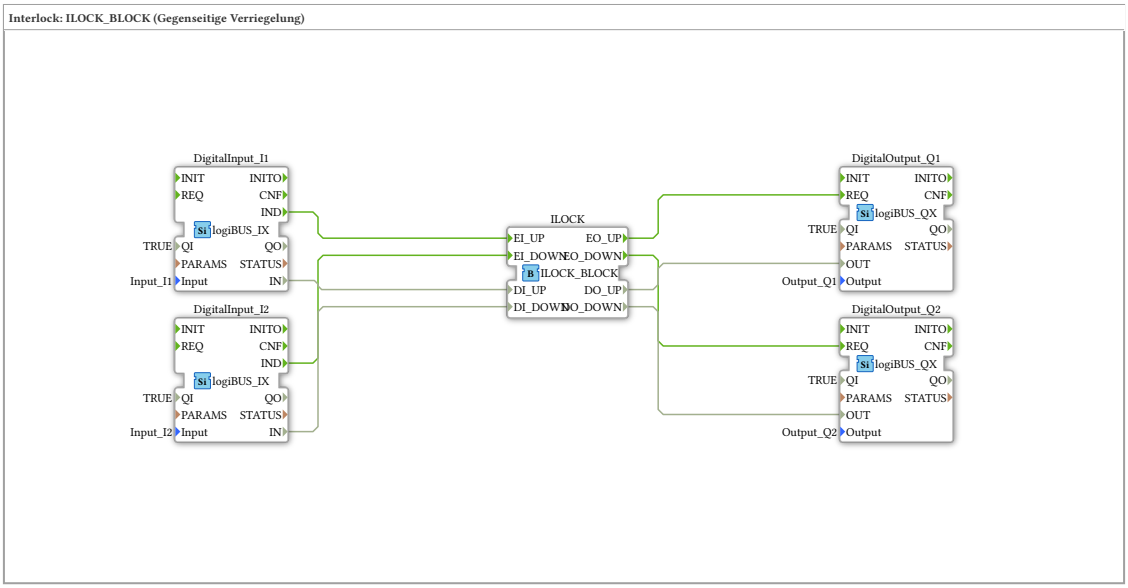
Lernziele:

- Verständnis des `E_REND`-Musters (Warten auf mehrere Ereignisse).
- Kombination von Ereignissteuerung und Zustandsspeicherung (Flip-Flop).
- Implementierung einer zentralen Reset-Logik.

Zusammenfassung

Die Übung 180 zeigt eine effektive Methode, um sicherzustellen, dass drei Bedingungen erfüllt (Ereignisse eingetreten) sein müssen, bevor ein Prozessschritt (Umschalten des Ausgangs) ausgeführt wird. Der `E_REND_3` Baustein fungiert hierbei als Ereignis-Sammler, während das `E_T_FF_SR` den aktuellen Status des Ausgangs speichert. Ein globaler Reset ermöglicht das Zurücksetzen der gesamten Logik in den Ausgangszustand.

2.2.1.357 Uebung_201: Interlock: ILOCK_BLOCK (Gegenseitige Verriegelung)



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung demonstriert die Realisierung einer **gegenseitigen Verriegelung** (Interlock) mittels des Funktionsbausteins `ILOCK_BLOCK`. Ziel ist es, zwei digitale Eingänge (`I1`, `I2`) so zu verknüpfen, dass immer nur einer der beiden zugehörigen Ausgänge (`Q1`, `Q2`) aktiv sein kann. Sobald ein Eingangssignal anliegt, wird der Ausgang gesetzt und der jeweils andere Ausgang gesperrt. Die Übung nutzt die Hardwarebausteine `logiBUS_IX` (Digitaleingang) und `logiBUS_QX` (Digitalausgang) sowie den Verriegelungsbaustein aus der `logiBUS`-Bibliothek.

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: `DigitalInput_I1`

- **Typ**: `logiBUS::io::DI::logiBUS_IX`
- **Verwendete interne FBs**: Keine (Hardwaretreiber)
  - **Parameter**:
    - `QI = TRUE`
    - `Input = Input_I1`
  - **Ereignisausgang**: `IND` (wird bei Signaländerung ausgelöst)
  - **Datenausgang**: `IN` (aktueller digitaler Wert)

Sub-Bausteine: `DigitalInput_I2`

- **Typ**: `logiBUS::io::DI::logiBUS_IX`
- **Verwendete interne FBs**: Keine
  - **Parameter**:
    - `QI = TRUE`
    - `Input = Input_I2`
  - **Ereignisausgang**: `IND`
  - **Datenausgang**: `IN`

Sub-Bausteine: `ILOCK`

- **Typ**: `logiBUS::signalprocessing::interlock::ILOCK_BLOCK`
- **Verwendete interne FBs**: Keine (vordefinierter Verriegelungsblock)
  - **Parameter**: Keine
  - **Ereigniseingänge**:
    - `EI_UP` – Ereignis für Kanal 1 (z. B. Taste „Auf“)
    - `EI_DOWN` – Ereignis für Kanal 2 (z. B. Taste „Ab“)
  - **Ereignisausgänge**:

    - EO_UP – Freigabe für Kanal 1
    - EO_DOWN – Freigabe für Kanal 2
  - **Dateneingänge**:
    - DI_UP – digitaler Wert für Kanal 1
    - DI_DOWN – digitaler Wert für Kanal 2
  - **Datenausgänge**:
    - DO_UP – gesetzter Ausgangswert für Kanal 1
    - DO_DOWN – gesetzter Ausgangswert für Kanal 2
- **Funktionsweise**: Der ILOCK_BLOCK wertet die beiden Kanäle aus. Wenn DI_UP aktiv (1) und das Ereignis EI_UP eintrifft, wird DO_UP auf 1 gesetzt und gleichzeitig DO_DOWN auf 0 zurückgesetzt (Verriegelung). Analog wird bei Aktivierung von Kanal 2 der Kanal 1 gesperrt. Es ist sichergestellt, dass nie beide Ausgänge gleichzeitig TRUE werden.

Sub-Bausteine: DigitalOutput_Q1

- **Typ**: logiBUS::io::DQ::logiBUS_QX
- **Verwendete interne FBs**: Keine
  - **Parameter**:
    - QI = TRUE
    - Output = Output_Q1
  - **Ereigniseingang**: REQ (Auslösung zur Ausgabe)
  - **Dateneingang**: OUT (auszugebender Wert)

Sub-Bausteine: DigitalOutput_Q2

- **Typ**: logiBUS::io::DQ::logiBUS_QX
- **Verwendete interne FBs**: Keine
  - **Parameter**:
    - QI = TRUE
    - Output = Output_Q2
  - **Ereigniseingang**: REQ
  - **Dateneingang**: OUT

Programmablauf und Verbindungen

Der Ablauf der Übung wird durch die **Ereignis- und Datenverbindungen** im SubApp-Netzwerk bestimmt:

1. **Eingangssignal-Erfassung** Die beiden Digitaleingänge DigitalInput_I1 und DigitalInput_I2 überwachen die physikalischen Eingänge Input_I1 bzw. Input_I2. Bei einer steigenden oder fallenden Flanke wird das Ereignis IND ausgelöst.
2. **Ereignisweiterleitung zum ILOCK**
   - DigitalInput_I1.IND ⟶ ILOCK.EI_UP
   - DigitalInput_I2.IND ⟶ ILOCK.EI_DOWN Gleichzeitig werden die aktuellen digitalen Werte über die Datenverbindungen an den ILOCK übergeben:
   - DigitalInput_I1.IN ⟶ ILOCK.DI_UP
   - DigitalInput_I2.IN ⟶ ILOCK.DI_DOWN
3. **Verriegelungslogik** Der ILOCK_BLOCK verarbeitet die eingehenden Ereignisse und Daten. Er setzt den Ausgang DO_UP (bzw. DO_DOWN) auf den Wert des zugehörigen Eingangs, sofern der andere Kanal nicht bereits aktiv ist. Durch die interne Logik wird sichergestellt, dass immer nur ein Kanal den Wert TRUE liefern kann. Die Ausgangsereignisse EO_UP und EO_DOWN werden entsprechend generiert.
4. **Ausgabe an die Hardware** Die Ereignisse und Daten des ILOCK werden an die Digitalausgänge weitergeleitet:
   - ILOCK.EO_UP ⟶ DigitalOutput_Q1.REQ
   - ILOCK.EO_DOWN ⟶ DigitalOutput_Q2.REQ
   - ILOCK.DO_UP ⟶ DigitalOutput_Q1.OUT
   - ILOCK.DO_DOWN ⟶ DigitalOutput_Q2.OUT

   Der jeweilige Ausgangsbaustein übernimmt den Wert und gibt ihn an den physikalischen Ausgang Output_Q1 bzw. Output_Q2 aus.

Zusammenfassende Funktionsweise: Wird der erste Eingang aktiviert (z. B. Taste an `Input_I1`), so schaltet der zugehörige Ausgang `Output_Q1` ein und der zweite Ausgang wird sofort ausgeschaltet. Wird anschließend der zweite Eingang aktiviert, wechselt der aktive Ausgang zu `Output_Q2` und `Output_Q1` wird wieder ausgeschaltet. Ein gleichzeitiges Halten beider Eingänge führt zu einer definierten Priorisierung (üblicherweise der zuletzt gedrückte Kanal).

Zusammenfassung

- **Ziel der Übung**: Grundlegendes Verständnis der gegenseitigen Verriegelung (Interlock) und deren Umsetzung mit dem vordefinierten Baustein `ILOCK_BLOCK`.
- **Schwierigkeitsgrad**: Einsteiger (Grundlagen der Ereignis- und Datenverarbeitung).
- **Vorkenntnisse**: Basiswissen über 4diac-IDE, Funktionsbausteine und logiBUS-Hardwaretreiber.
- **Lerninhalte**:
  - Umgang mit digitalen Ein-/Ausgangsbausteinen (`logiBUS_IX`, `logiBUS_QX`).
  - Ereignisgesteuerte Verbindungen zwischen Funktionsbausteinen.
  - Anwendung eines spezialisierten Verriegelungsblocks (`ILOCK_BLOCK`).
  - Debugging und Testen in der 4diac-IDE (z. B. durch Simulation mit Testtreibern).

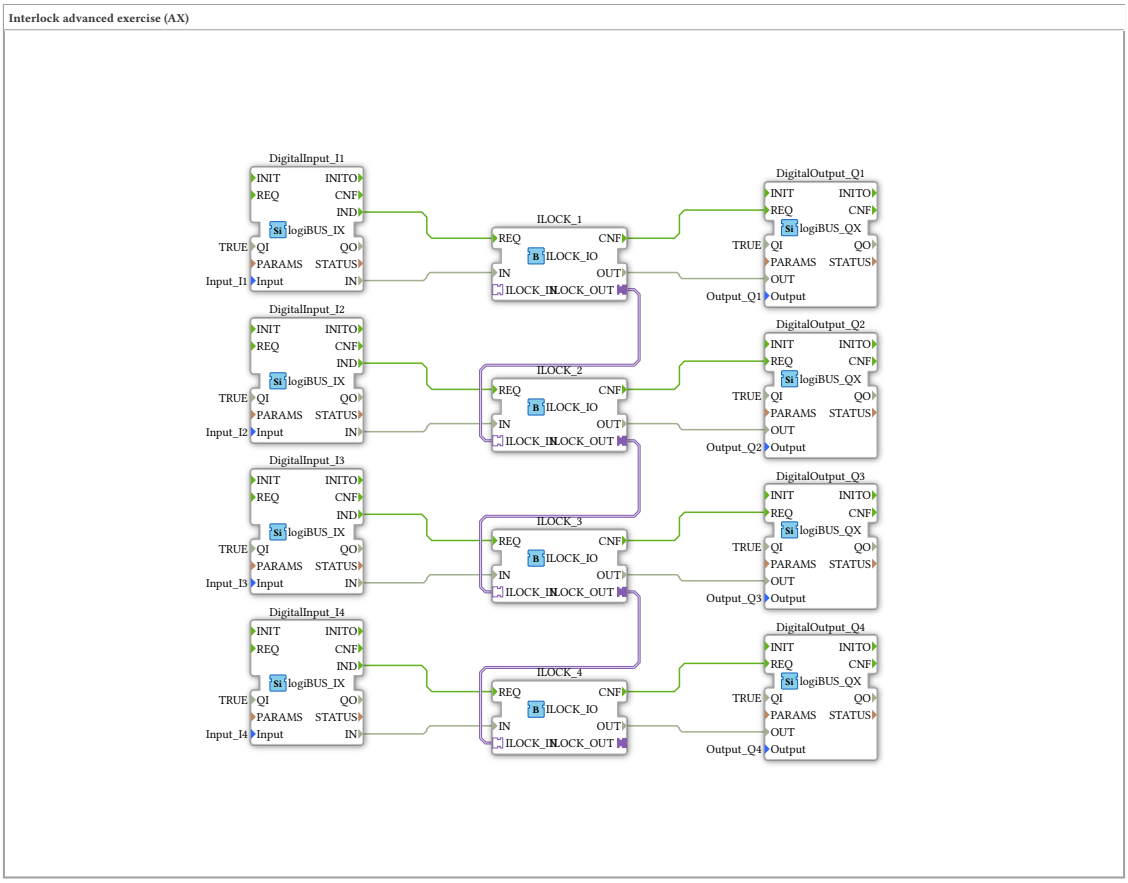
Nach Abschluss dieser Übung sind Sie in der Lage, einfache Interlock-Logiken in Steuerungsanwendungen zu integrieren und zu erweitern.

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

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

2.2.1.358 Uebung_201_Interlock: Interlock advanced exercise (AX)

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_201_Interlock (Interlock advanced exercise (AX)).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **Interlock advanced exercise (AX)**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_201_Interlock.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IX.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
- **DigitalOutput_Q4**: Instanz des Typs logiBUS::io::DQ::logiBUS_QX.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q4
- **ILOCK_1**: Instanz des Typs logiBUS::signalprocessing::interlock::ILOCK_IO.
- **DigitalInput_I2**: Instanz des Typs logiBUS::io::DI::logiBUS_IX.
  - Parameter QI = TRUE
  - Parameter Input = Input_I2
- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QX.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1
- **ILOCK_2**: Instanz des Typs logiBUS::signalprocessing::interlock::ILOCK_IO.
- **DigitalOutput_Q3**: Instanz des Typs logiBUS::io::DQ::logiBUS_QX.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q3
- **DigitalOutput_Q2**: Instanz des Typs logiBUS::io::DQ::logiBUS_QX.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q2
- **ILOCK_3**: Instanz des Typs logiBUS::signalprocessing::interlock::ILOCK_IO.
- **ILOCK_4**: Instanz des Typs logiBUS::signalprocessing::interlock::ILOCK_IO.
- **DigitalInput_I3**: Instanz des Typs logiBUS::io::DI::logiBUS_IX.
  - Parameter QI = TRUE
  - Parameter Input = Input_I3
- **DigitalInput_I4**: Instanz des Typs logiBUS::io::DI::logiBUS_IX.
  - Parameter QI = TRUE
  - Parameter Input = Input_I4

Verbindungen und Schnittstellen

Adapterverbindungen:

- ILOCK_1.ILOCK_OUT -> ILOCK_2.ILOCK_IN
- ILOCK_2.ILOCK_OUT -> ILOCK_3.ILOCK_IN
- ILOCK_3.ILOCK_OUT -> ILOCK_4.ILOCK_IN

Ereignisverbindungen:

- DigitalInput_I4.IND -> ILOCK_4.REQ
- DigitalInput_I3.IND -> ILOCK_3.REQ
- DigitalInput_I2.IND -> ILOCK_2.REQ
- DigitalInput_I1.IND -> ILOCK_1.REQ
- ILOCK_1.CNF -> DigitalOutput_Q1.REQ
- ILOCK_2.CNF -> DigitalOutput_Q2.REQ
- ILOCK_3.CNF -> DigitalOutput_Q3.REQ
- ILOCK_4.CNF -> DigitalOutput_Q4.REQ

Datenverbindungen:

- DigitalInput_I4.IN -> ILOCK_4.IN
- DigitalInput_I3.IN -> ILOCK_3.IN
- DigitalInput_I2.IN -> ILOCK_2.IN
- DigitalInput_I1.IN -> ILOCK_1.IN
- ILOCK_4.OUT -> DigitalOutput_Q4.OUT
- ILOCK_3.OUT -> DigitalOutput_Q3.OUT
- ILOCK_2.OUT -> DigitalOutput_Q2.OUT

- ILOCK_1.OUT -> DigitalOutput_Q1.OUT

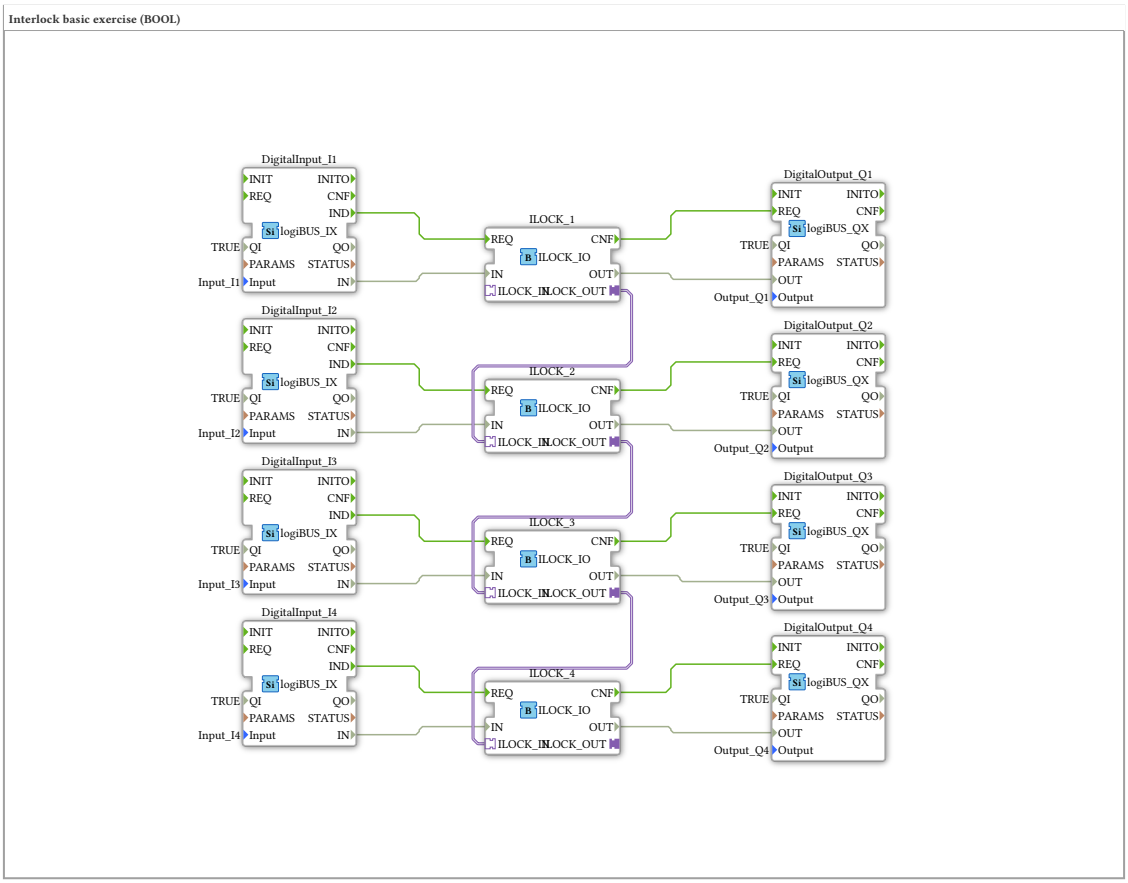
Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_201_Interlock demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.2.1.359 Uebung_201_Interlock_BOOL: Interlock basic exercise (BOOL)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung vermittelt die grundlegende Funktionsweise einer **Interlock-Schaltung** (Verriegelung) mit booleschen Signalen. Vier digitale Eingänge (`I1` bis `I4`) steuern über spezielle Interlock-Bausteine vier digitale Ausgänge (`Q1` bis `Q4`). Die Interlock-Blöcke sind in einer Kette angeordnet, sodass die Freigabe eines nachfolgenden Ausgangs erst dann erfolgen kann, wenn der vorherige Interlock-Block aktiv geschaltet wurde. So lässt sich eine sichere, sequenzielle Ablaufsteuerung realisieren.

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_I1 ... DigitalInput_I4** Typ: `logiBUS::io::DI::logiBUS_IX` – Ein digitaler Eingang des logiBUS-Systems.

- **DigitalOutput_Q1 ... DigitalOutput_Q4** Typ: `logiBUS::io::DQ::logiBUS_QX` – Ein digitaler Ausgang des logiBUS-Systems.
- **ILOCK_1 ... ILOCK_4** Typ: `logiBUS::signalprocessing::interlock::ILOCK_IO` – Spezieller Interlock-Funktionsbaustein, der den Zustand eines Eingangssignals nur dann an den Ausgang weitergibt, wenn die interne Verriegelungsbedingung erfüllt ist.

Es sind keine verschachtelten Sub-Bausteine (Unterapplikationen) vorhanden.

Programmablauf und Verbindungen

Ereignis- und Datenfluss

1. **Eingangsereignisse** Jeder Digitaleingang (z. B. `DigitalInput_I1`) erzeugt ein Ereignis (`IND`), sobald sich der Eingangswert ändert. Dieses Ereignis wird direkt an den zugehörigen Interlock-Block (z. B. `ILOCK_1.REQ`) gesendet.
2. **Datenweitergabe** Der Wert des Digitaleingangs (`IN`-Datenport) wird parallel zum Ereignis an den entsprechenden Interlock-Block (`ILOCK_x.IN`) übergeben.
3. **Verriegelungskette** Über Adapterverbindungen sind die Interlock-Blöcke kaskadiert:
   - `ILOCK_1.ILOCK_OUT` ⟶ `ILOCK_2.ILOCK_IN`
   - `ILOCK_2.ILOCK_OUT` ⟶ `ILOCK_3.ILOCK_IN`
   - `ILOCK_3.ILOCK_OUT` ⟶ `ILOCK_4.ILOCK_IN`

   Diese Verkettung bewirkt, dass ein Interlock-Baustein nur dann einen gültigen Ausgang liefert, wenn der vorherige Baustein ebenfalls aktiviert wurde.
4. **Ausgangssteuerung** Nach der internen Verarbeitung gibt jeder Interlock-Block ein Bestätigungsereignis (`CNF`) aus, das den zugehörigen Digitalausgang (z. B. `DigitalOutput_Q1.REQ`) ansteuert. Gleichzeitig wird der Datenwert (`OUT`) an den Ausgang übertragen.

Lernziele und Hinweise

- **Lernziel:** Verständnis der Interlock-Logik und der kaskadierten Freigabebedingungen.
- **Schwierigkeitsgrad:** Grundlegend.
- **Voraussetzungen:** Grundkenntnisse in der 4diac-IDE und im Umgang mit logiBUS-Bausteinen.
- **Start der Übung:** Legen Sie die Digitaleingänge (z. B. über Simulations-Testwerte) auf `TRUE` und beobachten Sie, wie die Ausgänge nacheinander aktiv werden.

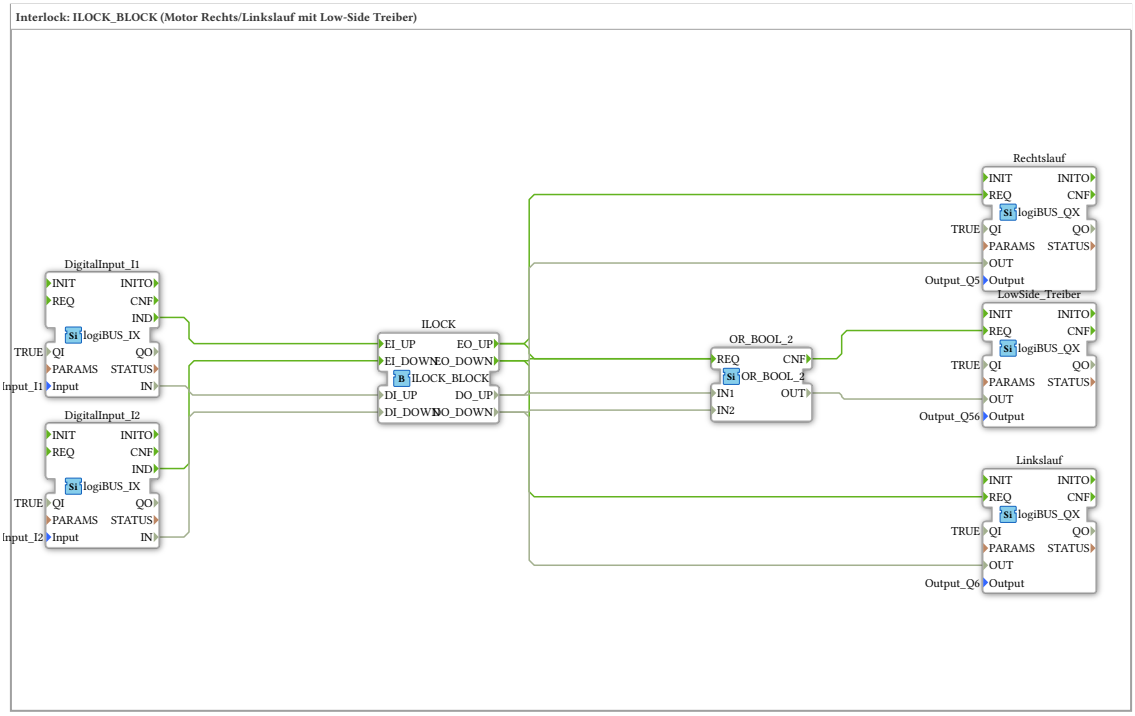
Zusammenfassung

Die Übung *Uebung_201_Interlock_BOOL* demonstriert den Aufbau einer einfachen Verriegelungskette mit vier Stufen. Jeder Eingang schaltet einen eigenen Interlock-Baustein, der seinen Ausgang nur dann freigibt, wenn die gesamte Kette bis zu ihm durchgängig aktiv ist. Die Realisierung erfolgt mit den logiBUS-Standardbausteinen für digitale Ein-/Ausgabe und dem speziellen Interlock-Funktionsbaustein `ILOCK_IO`. Dieses Grundprinzip lässt sich direkt auf sicherheitsrelevante Steuerungen (z. B. Shut-down-Systeme) übertragen.

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

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

2.2.1.360 Uebung_201b: Interlock: ILOCK_BLOCK (Motor Rechts/Linkslauf mit Low-Side Treiber)



• — ○ • — ○ • — ○ •

Einleitung

Diese Übung realisiert eine **Interlock-Schaltung** für einen Motor mit Rechts- und Linkslauf. Zusätzlich wird ein **Low-Side Treiber** angesteuert. Die Verriegelung (Interlock) verhindert, dass beide Richtungen gleichzeitig aktiv werden. Die Logik basiert auf dem speziellen Funktionsbaustein `ILOCK_BLOCK`.

Verwendete Funktionsbausteine (FBs)

Die SubApp verwendet folgende Funktionsbausteine:

Baustein-Name	Typ	Beschreibung
DigitalInput_I1	`logiBUS::io::DI::logiBUS_IX`	Digitaler Eingang für Sensor I1 (z. B. Taster „Auf“).
DigitalInput_I2	`logiBUS::io::DI::logiBUS_IX`	Digitaler Eingang für Sensor I2 (z. B. Taster „Ab“).
ILOCK	`logiBUS::signalprocessing::interlock::ILOCK_BLOCK`	Interlock-Baustein: verriegelt die beiden Richtungen gegeneinander.
Rechtslauf	`logiBUS::io::DQ::logiBUS_QX`	Digitaler Ausgang für den Rechtslauf (Q5).
Linkslauf	`logiBUS::io::DQ::logiBUS_QX`	Digitaler Ausgang für den Linkslauf (Q6).
LowSide_Treiber	`logiBUS::io::DQ::logiBUS_QX`	Digitaler Ausgang für den Low-Side Treiber (Q56).
OR_2_BOOL	`iec61131::bitwiseOperators::OR_2_BOOL`	Logisches ODER: aktiviert den Low-Side Treiber bei Rechts- oder Linkslauf.

Parameter der Instanzen

- **DigitalInput_I1**: `QI = TRUE`, `Input = Input_I1`

- **DigitalInput_I2**: `QI = TRUE`, `Input = Input_I2`
- **ILOCK**: keine Parameter gesetzt (Standardwerte)
- **Rechtslauf**: `QI = TRUE`, `Output = Output_Q5`
- **Linkslauf**: `QI = TRUE`, `Output = Output_Q6`
- **LowSide_Treiber**: `QI = TRUE`, `Output = Output_Q56`
- **OR_2_BOOL**: keine Parameter gesetzt

Programmablauf und Verbindungen

1. **Eingangssignale** Die digitalen Eingänge `DigitalInput_I1` und `DigitalInput_I2` lesen die physikalischen Signale der Taster oder Sensoren. Die Ereignisausgänge `.IND` lösen die entsprechenden Ereignisseingänge des Interlock-Bausteins aus:
   - `I1.IND` ⟶ `ILOCK.EI_UP`
   - `I2.IND` ⟶ `ILOCK.EI_DOWN`
2. **Interlock-Logik** Der Baustein `ILOCK` wertet die Daten-Eingänge `DI_UP` und `DI_DOWN` aus. Er stellt sicher, dass nie beide Ausgänge `DO_UP` und `DO_DOWN` gleichzeitig **TRUE** werden. Die Ereignisse `EO_UP` und `EO_DOWN` signalisieren, wenn eine Richtung aktiviert wird.
3. **Ausgangsansteuerung**
   - `ILOCK.EO_UP` und das Daten-Signal `DO_UP` steuern den Ausgang **Rechtslauf** (Q5).
   - `ILOCK.EO_DOWN` und `DO_DOWN` steuern den Ausgang **Linkslauf** (Q6).
   - Beide Ereignisse `EO_UP` und `EO_DOWN` sind mit dem ODER-Baustein `OR_2_BOOL` verbunden. Sobald eine Richtung aktiv ist, triggert der ODER-Baustein den **LowSide_Treiber** (Q56).
   - Gleichzeitig werden die Daten-Signale `DO_UP` und `DO_DOWN` auf die Eingänge `IN1` und `IN2` des ODER-Bausteins geführt. Der Ausgang `OR_2_BOOL.OUT` speist den Daten-Eingang des LowSide_Treibers.
4. **Zusammenhang** Der Low-Side Treiber wird nur dann eingeschaltet, wenn entweder Rechts- oder Linkslauf aktiv ist. Dadurch wird die Stromversorgung des Motors nur in diesen Zuständen freigegeben.

Lernziele:

- Verständnis einer Interlock-Schaltung für Motoren mit zwei Drehrichtungen.
- Ansteuerung eines Low-Side Treibers in Abhängigkeit einer ODER-Verknüpfung.
- Umgang mit den speziellen logiBUS-Funktionsbausteinen.

Benötigte Vorkenntnisse:

- Grundlagen der 4diac-IDE und des IEC 61499-Modells.
- Kenntnis über digitale Ein-/Ausgänge und boolesche Verknüpfungen.

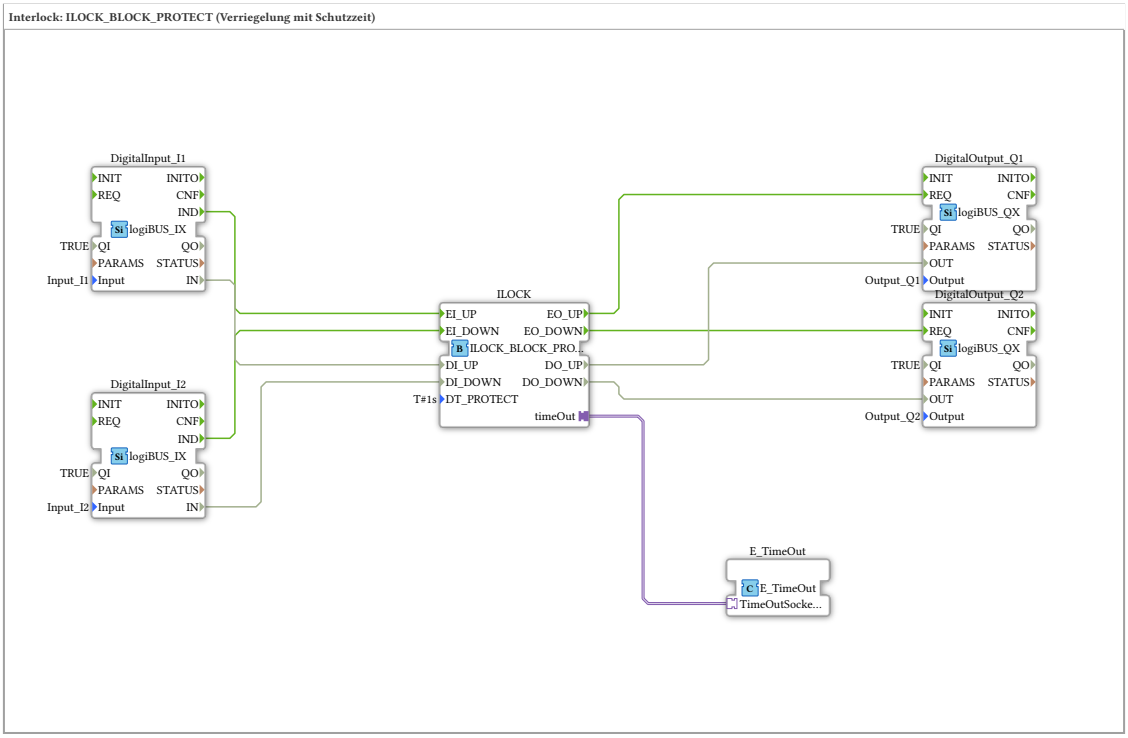
Zusammenfassung

Die Übung demonstriert eine sichere Ansteuerung eines Motors mit Rechts-/Linkslauf durch einen Interlock-Baustein. Ein Low-Side Treiber wird automatisch aktiviert, sobald eine der beiden Richtungen gewählt wird. Die gesamte Schaltung ist als SubApp realisiert und kann in übergeordneten Projekten wiederverwendet werden.

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

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

2.2.1.361 Uebung_202: Interlock: ILOCK_BLOCK_PROTECT (Verriegelung mit Schutzzeit)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert die Anwendung eines **Interlock-Bausteins mit Schutzzeit (ILOCK_BLOCK_PROTECT)**. Der Funktionsbaustein realisiert eine Verriegelung zwischen zwei gegenläufigen Bewegungen (z. B. Auf/Ab eines Antriebs) und verhindert durch eine einstellbare Schutzzeit ein sofortiges Umschalten der Richtung. Die logischen Eingangssignale werden über digitale Eingänge eingelesen und die Ausgangssignale über digitale Ausgänge ausgegeben. Ein Zeitgeberbaustein ist als Adapter an den Interlock angebunden, um das Zeitverhalten zu steuern.

Verwendete Funktionsbausteine (FBs)

Bausteinname	Typ	Beschreibung
DigitalInput_I1	`logiBUS::io::DI::logiBUS_IX`	Digitaler Eingang – liest das Signal von **Input_I1** (z. B. Taster „Auf“)
DigitalInput_I2	`logiBUS::io::DI::logiBUS_IX`	Digitaler Eingang – liest das Signal von **Input_I2** (z. B. Taster „Ab“)
ILOCK	`logiBUS::signalprocessing::interlock::ILOCK_BLOCK_PROTECT`	Interlock-Baustein mit Schutzzeit – verriegelt die Ausgänge und erzwingt eine minimale Umschaltverzögerung (Parameter `DT_PROTECT = T#1s`)
DigitalOutput_Q1	`logiBUS::io::DQ::logiBUS_QX`	Digitaler Ausgang – steuert **Output_Q1** (z. B. Relais „Auf“)
DigitalOutput_Q2	`logiBUS::io::DQ::logiBUS_QX`	Digitaler Ausgang – steuert **Output_Q2** (z. B. Relais „Ab“)
E_TimeOut	`iec61499::events::E_TimeOut`	Ereignisgesteuerter Zeitgeber – liefert die Zeitbasis für die Schutzzeit (als Adapter verbunden)

Parameter der FBs

- **DigitalInput_I1**: `QI = TRUE`, `Input = Input_I1`
- **DigitalInput_I2**: `QI = TRUE`, `Input = Input_I2`
- **ILOCK**: `DT_PROTECT = T#1s` (Schutzzeit 1 Sekunde)
- **DigitalOutput_Q1**: `QI = TRUE`, `Output = Output_Q1`
- **DigitalOutput_Q2**: `QI = TRUE`, `Output = Output_Q2`

Programmablauf und Verbindungen

Ereignisverbindungen

1. **DigitalInput_I1.IND → ILOCK.EI_UP**
   - Eine steigende Flanke am Eingang I1 löst das Ereignis „Auf“ beim Interlock aus.
2. **DigitalInput_I2.IND → ILOCK.EI_DOWN**
   - Eine steigende Flanke am Eingang I2 löst das Ereignis „Ab“ aus.
3. **ILOCK.EO_UP → DigitalOutput_Q1.REQ**
   - Wenn der Interlock den Zustand „Auf“ freigibt, wird der digitale Ausgang Q1 gesetzt.
4. **ILOCK.EO_DOWN → DigitalOutput_Q2.REQ**
   - Wenn der Interlock den Zustand „Ab“ freigibt, wird der digitale Ausgang Q2 gesetzt.

Datenverbindungen

- **DigitalInput_I1.IN → ILOCK.DI_UP**
- **DigitalInput_I2.IN → ILOCK.DI_DOWN**
- **ILOCK.DO_UP → DigitalOutput_Q1.OUT**
- **ILOCK.DO_DOWN → DigitalOutput_Q2.OUT**

Adapterverbindung

- **ILOCK.timeOut ↔ E_TimeOut.TimeOutSocket**
  - Der Zeitgeber wird als Adapter an den Interlock angeschlossen und stellt die benötigte Zeitbasis für die Schutzzeit zur Verfügung.

Funktionsweise

- Wird der Eingang **I1** aktiv, sendet der Interlock nach Ablauf der Schutzzeit das Ereignis **EO_UP** und setzt **DO_UP**.
- Wird direkt danach **I2** aktiv, wird der Umschaltvorgang blockiert, bis die Schutzzeit abgelaufen ist.
- Der Parameter `DT_PROTECT` definiert die Mindestzeit zwischen zwei Richtungswechseln – hier 1 Sekunde.
- Die digitalen Eingänge werden mit ihren logischen Zuständen (IN) verarbeitet, während die Ereignisse (IND) die Zustandsänderung des Interlocks triggern.

Lernziele dieser Übung

- Verständnis der Funktionsweise eines Interlock-Bausteins mit Schutzzeit
- Anwendung des Bausteins `ILOCK_BLOCK_PROTECT`
- Einbindung eines Zeitgebers als Adapter
- Zusammenhang zwischen Ereignis- und Datenflüssen in 4diac-IDE

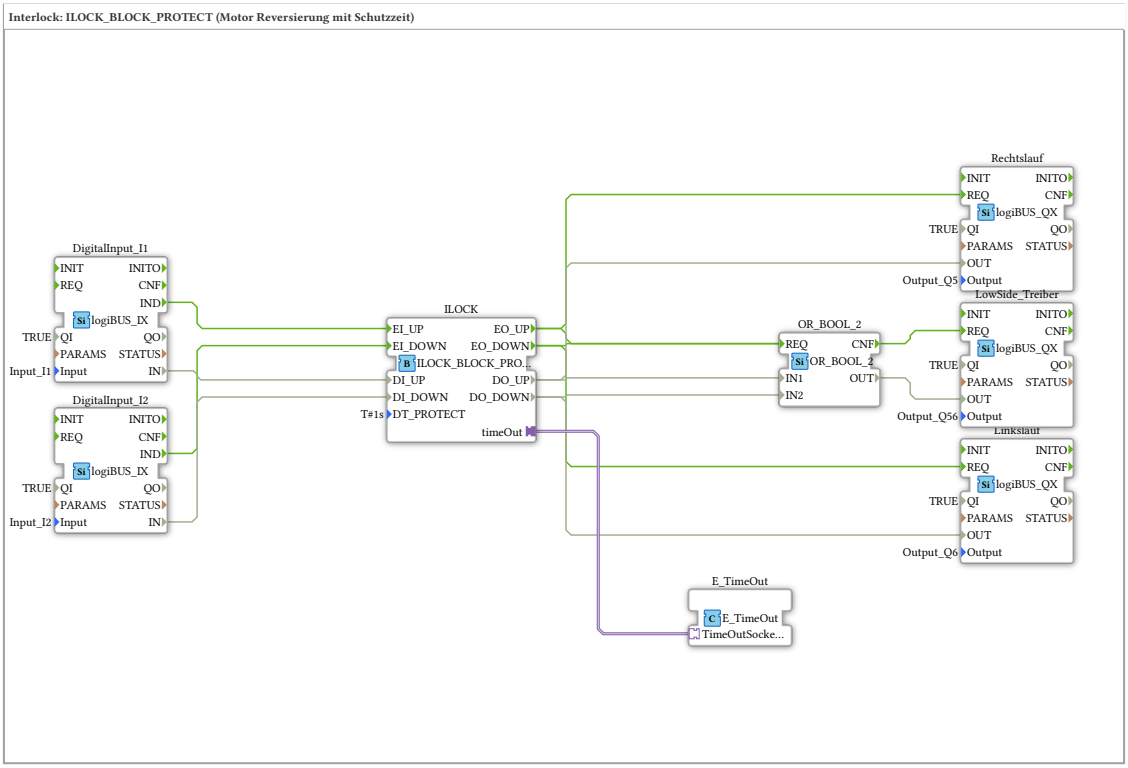
Zusammenfassung

Die Übung **Uebung_202** veranschaulicht eine einfache Verriegelung mit Schutzzeit. Zwei Taster (Auf/Ab) steuern über einen Interlock-Baustein zwei Ausgänge. Der integrierte Zeitschutz verhindert ein sofortiges Umschalten und schützt damit mechanische Komponenten. Die Implementierung nutzt digitale Ein-/Ausgänge und einen externen Zeitgeber, der als Adapter an den Interlock angebunden wird. Dieses Grundprinzip ist in der Automatisierungstechnik (z. B. bei Hubwerken oder Schiebetoren) weit verbreitet.

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

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

2.2.1.362 Uebung_202b: Interlock: ILOCK_BLOCK_PROTECT (Motor Reversierung mit Schutzzeit)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung implementiert eine Motorreversierung mit Interlock und Schutzzeit (ILOCK_BLOCK_PROTECT). Über zwei digitale Eingänge wird die Drehrichtung (Rechts- oder Linkslauf) angesteuert, wobei der Interlock-Block verhindert, dass beide Ausgänge gleichzeitig aktiv werden. Ein Low-Side-Treiber wird über eine ODER-Verknüpfung in beiden Fällen mitgeschaltet. Zusätzlich ist eine Schutzzeit (DT_PROTECT = 1 s) integriert, die einen Richtungswechsel erst nach Ablauf dieser Zeit erlaubt.

Verwendete Funktionsbausteine (FBs)

Die Übung besteht aus folgenden Funktionsbausteinen (FB-Instanzen), die alle in der SubApp `Uebung_202b` enthalten sind.

- **DigitalInput_I1**: Typ `logiBUS::io::DI::logiBUS_IX` Lest den digitalen Eingang `Input_I1` ein.
  - Parameter: QI = TRUE, Input = `Input_I1`
  - Ereignisausgang: IND (Indication)
  - Datenausgang: IN (Wert des Eingangs)
- **DigitalInput_I2**: Typ `logiBUS::io::DI::logiBUS_IX` Lest den digitalen Eingang `Input_I2` ein.
  - Parameter: QI = TRUE, Input = `Input_I2`
  - Ereignisausgang: IND
  - Datenausgang: IN
- **ILOCK**: Typ `logiBUS::signalprocessing::interlock::ILOCK_BLOCK_PROTECT` Kernbaustein der Interlock-Funktion mit Schutzzeit.
  - Parameter: DT_PROTECT = T#1s (Schutzzeit 1 Sekunde)
  - Ereigniseingänge: EI_UP, EI_DOWN
  - Ereignisausgänge: EO_UP, EO_DOWN
  - Dateneingänge: DI_UP, DI_DOWN
  - Datenausgänge: DO_UP, DO_DOWN

  - Adapterschnittstelle: timeOut (verbunden mit E_TimeOut)

- **Rechtslauf**: Typ `logiBUS::io::DQ::logiBUS_QX` Steuert den digitalen Ausgang `Output_Q5` (Rechtslauf).
  - Parameter: QI = TRUE, Output = `Output_Q5`
  - Ereigniseingang: REQ
  - Dateneingang: OUT
- **Linkslauf**: Typ `logiBUS::io::DQ::logiBUS_QX` Steuert den digitalen Ausgang `Output_Q6` (Linkslauf).
  - Parameter: QI = TRUE, Output = `Output_Q6`
  - Ereigniseingang: REQ
  - Dateneingang: OUT
- **LowSide_Treiber**: Typ `logiBUS::io::DQ::logiBUS_QX` Steuert den gemeinsamen Low-Side-Ausgang `Output_Q56`.
  - Parameter: QI = TRUE, Output = `Output_Q56`
  - Ereigniseingang: REQ
  - Dateneingang: OUT
- **OR_2_BOOL**: Typ `iec61131::bitwiseOperators::OR_2_BOOL` Logische ODER-Verknüpfung zweier Boolescher Werte.
  - Ereigniseingang: REQ
  - Ereignisausgang: CNF
  - Dateneingänge: IN1, IN2
  - Datenausgang: OUT
- **E_TimeOut**: Typ `iec61499::events::E_TimeOut` Zeitgeber, der die Schutzzeit des ILOCK überwacht.
  - Adapterschnittstelle: TimeOutSocket (verbunden mit ILOCK.timeOut)

Programmablauf und Verbindungen

Die Komponenten sind wie folgt verschaltet:

1. **Eingangssignale**:
   - `DigitalInput_I1` (Taster/Sensor für Rechtslauf) sendet über seinen Ereignisausgang IND ein Ereignis an den Ereigniseingang `EI_UP` des ILOCK. Gleichzeitig wird der Datenwert `IN` an den Dateneingang `DI_UP` übergeben.
   - `DigitalInput_I2` (Taster/Sensor für Linkslauf) sendet analog an `EI_DOWN` und `DI_DOWN`.
2. **Interlock mit Schutzzeit (ILOCK)**: Der ILOCK-Block wertet die Eingangssignale aus. Bei einem gültigen Befehl (z. B. DI_UP = TRUE und Ereignis an EI_UP) wird der entsprechende Ausgang (DO_UP) aktiviert und ein Ereignis an `EO_UP` ausgegeben. Gleichzeitig wird der andere Ausgang (DO_DOWN) deaktiviert. Die Schutzzeit `DT_PROTECT = 1s` verhindert einen sofortigen Richtungswechsel; erst nach Ablauf der Zeit darf die Gegenrichtung angenommen werden. Der Adapter `timeOut` ist mit dem `E_TimeOut`-Baustein verbunden, der die Zeitüberwachung realisiert.
3. **Ausgangsansteuerung**:
   - Das Ereignis `EO_UP` des ILOCK triggert den `Rechtslauf`-Baustein (Eingang REQ) und übergibt den Datenwert `DO_UP` an den Dateneingang OUT. Somit wird der Ausgang `Output_Q5` gesetzt.
   - Analog wird bei `EO_DOWN` der `Linkslauf`-Baustein aktiviert und `Output_Q6` gesetzt.
   - Beide Ereignisse (EO_UP und EO_DOWN) werden auch an den Ereigniseingang REQ des `OR_2_BOOL`-Bausteins weitergeleitet. Der Datenwertausgang `DO_UP` geht auf IN1, `DO_DOWN` auf IN2 des ODER-Bausteins. Der Ausgang OUT des ODER-Bausteins wird an den `LowSide_Treiber` übergeben, sodass der gemeinsame Ausgang `Output_Q56` immer dann aktiv ist, wenn entweder Rechts- oder Linkslauf aktiv ist.
4. **Zeitüberwachung**: Der `E_TimeOut`-Baustein ist über eine Adapterverbindung mit dem ILOCK verbunden und stellt die notwendige Timing-Funktionalität für die Schutzzeit bereit.

Zusammenfassend ergibt sich folgender Ablauf:

- Betätigung von Eingang I1 ⟶ Rechtslauf wird eingeschaltet, Linkslauf ist gesperrt, Low-Side-Treiber wird aktiv.

- Bei Wechsel auf I2: die Schutzzeit von 1s muss abgelaufen sein, bevor der Linkslauf aktiv wird. Während der Schutzzeit bleibt der letzte Zustand erhalten.
- Der Low-Side-Treiber folgt dem aktuell aktiven Laufbefehl.

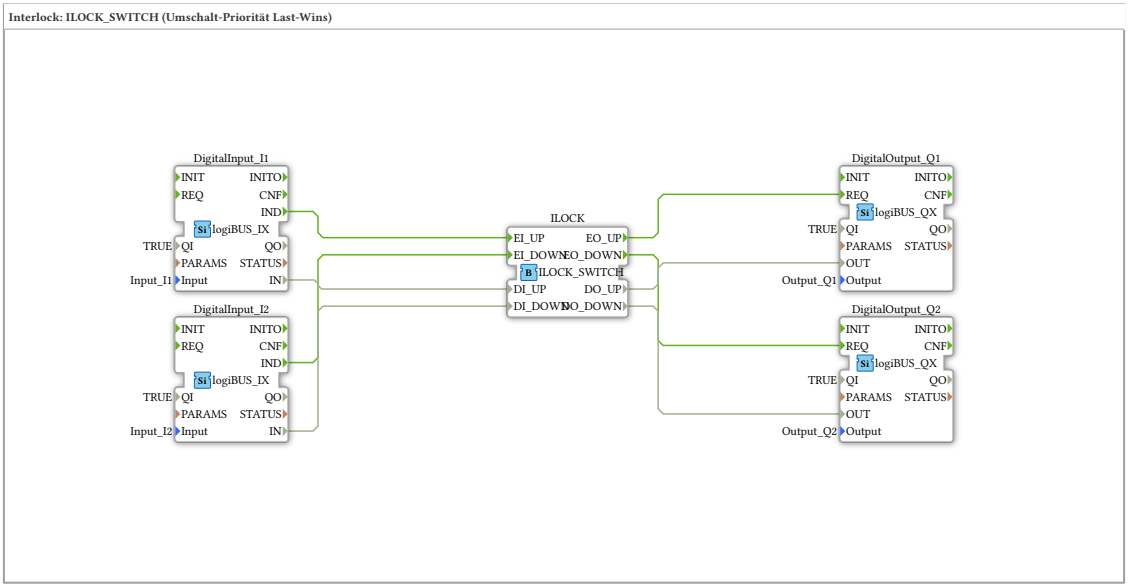
Zusammenfassung

Die Übung `Uebung_202b` vermittelt den Einsatz des Interlock-Bausteins `ILOCK_BLOCK_PROTECT` zur sicheren Motorreversierung. Durch die Integration einer Schutzzeit wird ein zu schneller Richtungswechsel verhindert, was Bauteilschäden vorbeugt. Die Verknüpfung mit einem ODER-Gatter zur gemeinsamen Low-Side-Ansteuerung sowie die klare Trennung von Ereignis- und Datenflüssen verdeutlichen die typische Struktur einer 4diac-IDE-Steuerung für logiBUS-Hardware.

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

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

2.2.1.363 Uebung_203: Interlock: ILOCK_SWITCH (Umschalt-Priorität Last-Wins)



Interlock: ILOCK_SWITCH (Umschalt-Priorität Last-Wins)

• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert den Einsatz eines Interlock-Bausteins vom Typ `ILOCK_SWITCH` zur Priorisierung zweier konkurrierender Anforderungen. Die Schaltung realisiert eine **Umschalt-Priorität nach dem Last-Wins-Prinzip**: Der zuletzt aktivierte Eingang erhält die Freigabe (Priorität). Sobald beide Eingänge wieder inaktiv sind, wird der Ausgang zurückgesetzt.

Zwei digitale Eingangssignale (I1, I2) steuern über den Interlock-Baustein zwei separate digitale Ausgänge (Q1, Q2). Die Logik verhindert, dass beide Ausgänge gleichzeitig aktiv sind, und stellt sicher, dass nur der zuletzt betätigte Kanal durchgeschaltet wird.

Verwendete Funktionsbausteine (FBs)

Die SubApp enthält folgende Funktionsbausteine:

- **DigitalInput_I1** (Typ: `logiBUS::io::DI::logiBUS_IX`) *Parameter*: `QI = TRUE`, `Input = Input_I1` *Ereignisausgang*: `IND` (aktiviert bei steigender Flanke des Eingangs) *Datenausgang*: `IN` (aktueller Eingangswert)
- **DigitalInput_I2** (Typ: `logiBUS::io::DI::logiBUS_IX`) *Parameter*: `QI = TRUE`, `Input = Input_I2`

- **ILOCK** (Typ: `logiBUS::signalprocessing::interlock::ILOCK_SWITCH`) Keine Parameter. *Ereigniseingänge*: `EI_UP`, `EI_DOWN` *Dateneingänge*: `DI_UP`, `DI_DOWN` *Ereignisausgänge*: `EO_UP`, `EO_DOWN` *Datenausgänge*: `DO_UP`, `DO_DOWN`
- **DigitalOutput_Q1** (Typ: `logiBUS::io::DQ::logiBUS_QX`) *Parameter*: `QI = TRUE`, `Output = Output_Q1` *Ereigniseingang*: `REQ` (aktiviert die Ausgabe) *Dateneingang*: `OUT` (zu setzender Wert)
- **DigitalOutput_Q2** (Typ: `logiBUS::io::DQ::logiBUS_QX`) *Parameter*: `QI = TRUE`, `Output = Output_Q2`

Programmablauf und Verbindungen

Die Schaltung arbeitet **ereignisgesteuert** und nutzt sowohl Ereignis- als auch Datenverbindungen.

1. **Ereignissteuerung**:
   - Eine steigende Flanke am Eingang `I1` erzeugt am `DigitalInput_I1` ein Ereignis an `IND`, das mit dem Ereigniseingang `EI_UP` des Interlock-Bausteins verbunden ist.
   - Analog wird bei einer Flanke an `I2` das Ereignis an `EI_DOWN` gesendet.
   - Der Interlock-Baustein entscheidet anhand der Priorität (Last-Wins), welcher Kanal freigegeben wird, und löst entweder `EO_UP` oder `EO_DOWN` aus.
   - Das Ereignis `EO_UP` triggert den Digitalausgang `Q1` über `REQ`, während `EO_DOWN` den Ausgang `Q2` triggert.
2. **Datenfluss**:
   - Der aktuelle Wert von `I1` (über `IN` von `DigitalInput_I1`) wird an den Dateneingang `DI_UP` des Interlocks übergeben.
   - Der Wert von `I2` wird an `DI_DOWN` übergeben.
   - Der Interlock-Baustein gibt den freigegebenen Kanal über die entsprechenden Datenausgänge weiter (`DO_UP` ⟶ `OUT` von `DigitalOutput_Q1`, `DO_DOWN` ⟶ `OUT` von `DigitalOutput_Q2`).
3. **Funktionsweise des `ILOCK_SWITCH`**:
   - Bei gleichzeitiger oder kollidierender Anforderung gewinnt der zuletzt eingetroffene Impuls (Last-Wins).
   - Nur einer der beiden Ausgänge kann aktiv sein.
   - Sobald beide Eingänge auf `FALSE` zurückgehen, werden auch die Ausgänge zurückgesetzt (sofern der Baustein entsprechend konfiguriert ist).

Lernziele dieser Übung:

- Verständnis des Interlock-Prinzips und der Last-Wins-Priorität.
- Umgang mit ereignisgesteuerten Funktionsbausteinen in 4diac IDE.
- Verknüpfung von digitalen Ein-/Ausgängen mit einem Interlock-Baustein.
- Fehlersuche und Verhaltenstests durch Simulation.

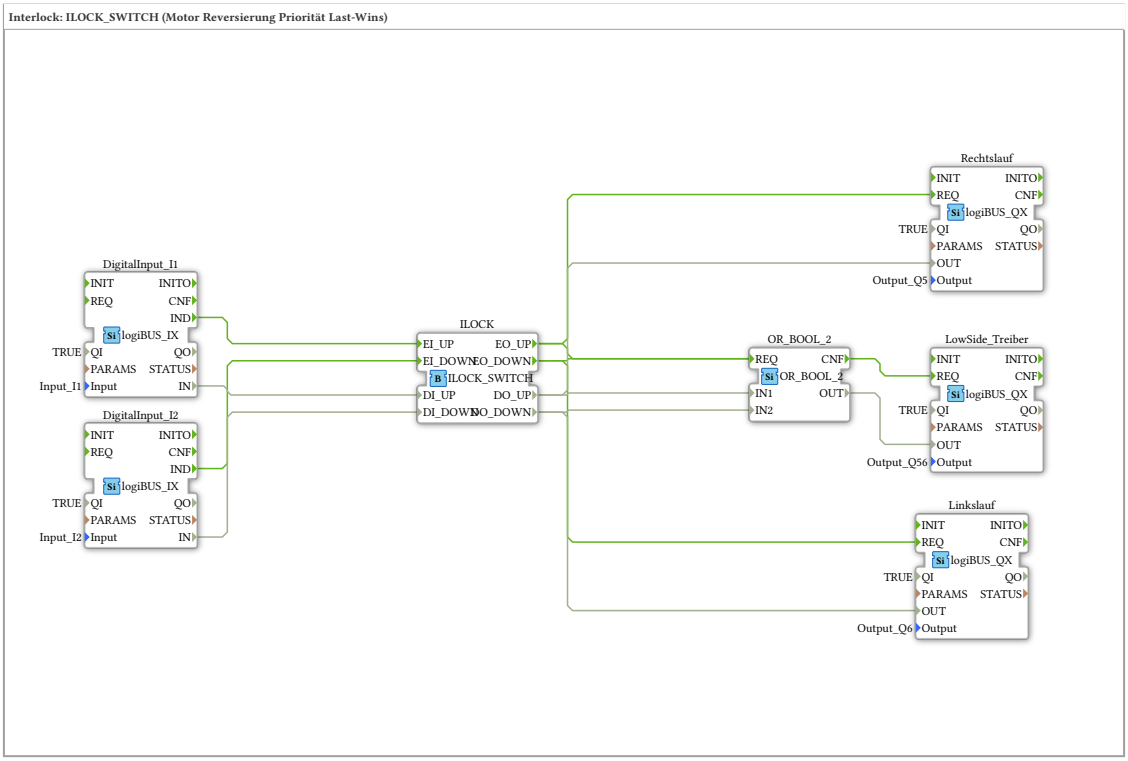
Zusammenfassung

Die Übung `Uebung_203` setzt einen `ILOCK_SWITCH`-Baustein ein, um zwei konkurrierende digitale Eingänge zu priorisieren. Die Last-Wins-Logik stellt sicher, dass stets nur der zuletzt aktivierte Kanal durchgeschaltet wird. Die Implementierung erfolgt vollständig ereignisgesteuert mit entsprechenden Datenverbindungen, was eine saubere und deterministische Steuerung ermöglicht. Diese Grundschaltung ist ein typischer Baustein für Sicherheits- und Verriegelungslogiken in der Automatisierungstechnik.

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

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

2.2.1.364 Uebung_203b: Interlock: ILOCK_SWITCH (Motor Reversierung Priorität Last-Wins)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert den Einsatz eines **ILOCK_SWITCH**-Funktionsbausteins zur sicheren Ansteuerung eines Motors mit Reversierfunktion. Das Prinzip der **Last-Wins-Priorität** sorgt dafür, dass bei gleichzeitig anliegenden Steuersignalen das zuletzt aktive Signal Vorrang hat – ein Kurzschluss durch gleichzeitige Aktivierung beider Drehrichtungen wird verhindert. Ein zusätzlicher **Low-Side-Treiber** wird bei jeder aktiven Drehrichtung eingeschaltet, um die Last (z. B. Motor) mit Spannung zu versorgen.

Verwendete Funktionsbausteine (FBs)

Digitaleingänge

- **DigitalInput_I1** (Typ: `logiBUS::io::DI::logiBUS_IX`)
  - Parameter: `QI = TRUE`, `Input = Input_I1`
  - Ereignisausgang: `IND`
  - Datenausgang: `IN`
- **DigitalInput_I2** (Typ: `logiBUS::io::DI::logiBUS_IX`)
  - Parameter: `QI = TRUE`, `Input = Input_I2`
  - Ereignisausgang: `IND`
  - Datenausgang: `IN`

Interlock-Baustein

- **ILOCK** (Typ: `logiBUS::signalprocessing::interlock::ILOCK_SWITCH`)
  - Ereigniseingänge: `EI_UP`, `EI_DOWN`
  - Dateneingänge: `DI_UP`, `DI_DOWN`
  - Ereignisausgänge: `EO_UP`, `EO_DOWN`
  - Datenausgänge: `DO_UP`, `DO_DOWN`

Digitalausgänge

- **Rechtslauf** (Typ: `logiBUS::io::DQ::logiBUS_QX`)

  - Parameter: `QI = TRUE`, `Output = Output_Q5`
  - Eingang: `REQ`, Daten: `OUT`
- **Linkslauf** (Typ: `logiBUS::io::DQ::logiBUS_QX`)
  - Parameter: `QI = TRUE`, `Output = Output_Q6`
  - Eingang: `REQ`, Daten: `OUT`
- **LowSide_Treiber** (Typ: `logiBUS::io::DQ::logiBUS_QX`)
  - Parameter: `QI = TRUE`, `Output = Output_Q56`
  - Eingang: `REQ`, Daten: `OUT`

Logikgatter

- **OR_2_BOOL** (Typ: `iec61131::bitwiseOperators::OR_2_BOOL`)
  - Ereigniseingang: `REQ`
  - Dateneingänge: `IN1`, `IN2`
  - Ereignisausgang: `CNF`
  - Datenausgang: `OUT`

Programmablauf und Verbindungen

1. Die beiden digitalen Eingänge **Input_I1** und **Input_I2** liefern über die Bausteine `DigitalInput_I1` bzw. `DigitalInput_I2` die Steuersignale für die Drehrichtung (z. B. Taster für Rechts- und Linkslauf).
2. Die Ereignisse `IND` sowie die Datenwerte `IN` werden an den Interlock-Baustein **ILOCK** weitergeleitet:
   - `DigitalInput_I1.IND` ⟶ `ILOCK.EI_UP` | `DigitalInput_I1.IN` ⟶ `ILOCK.DI_UP`
   - `DigitalInput_I2.IND` ⟶ `ILOCK.EI_DOWN` | `DigitalInput_I2.IN` ⟶ `ILOCK.DI_DOWN`
3. Der `ILOCK_SWITCH` entscheidet nach der **Last-Wins**-Logik, welcher Ausgang aktiviert wird:
   - Bei einem Ereignis an `EI_UP` wird `DO_UP = DI_UP` gesetzt und `DO_DOWN` zurückgesetzt (sofern beide Eingänge aktiv).
   - Bei einem Ereignis an `EI_DOWN` wird `DO_DOWN = DI_DOWN` gesetzt und `DO_UP` zurückgesetzt.
   - Die entsprechenden Ereignisausgänge (`EO_UP`, `EO_DOWN`) triggern die nachfolgenden Bausteine.
4. Der Ausgang `DO_UP` wird auf den Datenausgang **Rechtslauf** (`OUT`) gegeben und dessen `REQ`-Ereignis über `EO_UP` aktiviert. Gleiches gilt für `DO_DOWN` und **Linkslauf**.
5. Die Signale `DO_UP` und `DO_DOWN` werden gleichzeitig dem ODER-Gatter **OR_2_BOOL** zugeführt:
   - `ILOCK.DO_UP` ⟶ `OR_2_BOOL.IN1`
   - `ILOCK.DO_DOWN` ⟶ `OR_2_BOOL.IN2`
   - Die Ereignisse `EO_UP` und `EO_DOWN` werden auf `OR_2_BOOL.REQ` zusammengeführt.
6. Sobald mindestens einer der beiden Datenwerte `TRUE` ist, liefert `OR_2_BOOL.OUT = TRUE`. Das Bestätigungsereignis `CNF` aktiviert dann den **LowSide_Treiber** (`REQ`), der die gemeinsame Versorgung der Last (z. B. Motorspannung) über `Output_Q56` einschaltet.

Durch diese Verschaltung wird sichergestellt:

- Nie beide Drehrichtungen gleichzeitig aktiv.
- Die Last wird nur bestromt, wenn eine Drehrichtung angefordert wird.
- Die **Last-Wins-Priorität** verhindert Blockaden bei gleichzeitigen Tasterbetätigungen.

Zusammenfassung

Die Übung vermittelt die sichere Ansteuerung eines Reversiermotors mittels Interlock-Baustein und Last-Wins-Logik. **Lernziele:**

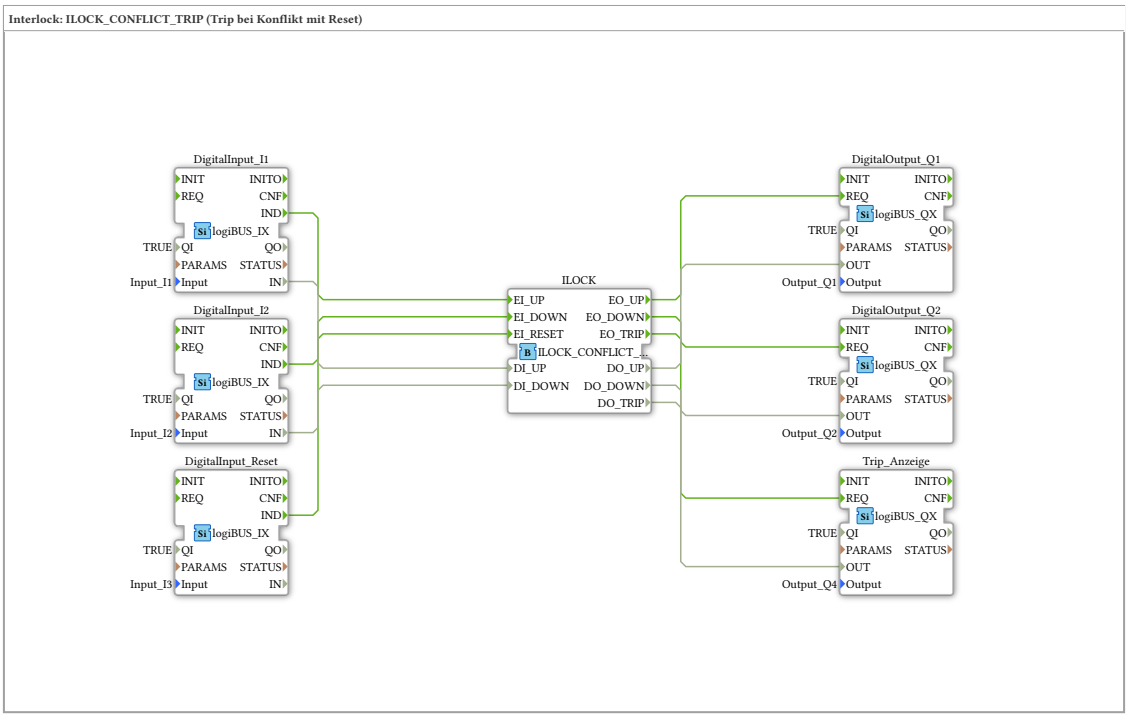
- Verständnis von Interlock-Mechanismen zur Vermeidung von Kurzschlüssen.
- Umgang mit dem ILOCK_SWITCH-Funktionsbaustein (Ereignis-/Datenschnittstelle).
- Kombination von Ereignis- und Datenflüssen in 4diac-IDE.
- Realisierung einer Hilfsspannung (Low-Side-Treiber) in Abhängigkeit der Motoranforderung.

Schwierigkeitsgrad: Mittel **Vorkenntnisse:** Grundlagen der 4diac-IDE, Ereignis- und Datenverbindungen, einfache Logikgatter. **Start der Übung:** Öffnen Sie die Subapplikation `Uebung_203b` und simulieren Sie die digitalen Eingänge `Input_I1` / `Input_I2`. Beobachten Sie die Ausgänge `Output_Q5` (Rechts), `Output_Q6` (Links) und `Output_Q56` (Low-Side).

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

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

2.2.1.365 Uebung_204: Interlock: ILOCK_CONFLICT_TRIP (Trip bei Konflikt mit Reset)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung realisiert eine Interlock-Funktion mit Konflikterkennung und Trip-Auslösung, die durch einen Reset zurückgesetzt werden kann. Sie zeigt den typischen Einsatz eines Interlock-Bausteins zur Vermeidung gleichzeitiger, widersprüchlicher Ansteuerungen (z. B. Auf/Ab-Bewegung). Der ILOCK_CONFLICT_TRIP erzeugt bei einem Konflikt einen Trip (Störausgang) und sperrt die Ausgänge, bis ein explizites Rücksetzsignal anliegt.

Verwendete Funktionsbausteine (FBs)

Die Übung besteht aus folgenden Funktionsbausteinen, die über Ereignis- und Datenleitungen verbunden sind:

- **DigitalInput_I1** – Typ: `logiBUS::io::DI::logiBUS_IX` Parameter: `QI = TRUE`, `Input = Input_I1` Digitaleingang für das Auf-Signal (UP).
- **DigitalInput_I2** – Typ: `logiBUS::io::DI::logiBUS_IX` Parameter: `QI = TRUE`, `Input = Input_I2` Digitaleingang für das Ab-Signal (DOWN).
- **DigitalInput_Reset** – Typ: `logiBUS::io::DI::logiBUS_IX` Parameter: `QI = TRUE`, `Input = Input_I3` Digitaleingang für das Rücksetzsignal.
- **ILOCK** – Typ: `logiBUS::signalprocessing::interlock::ILOCK_CONFLICT_TRIP` Der zentrale Interlock-Baustein. Er wertet die Eingangssignale aus und steuert die Ausgänge. Bei gleichzeitig anliegendem UP und DOWN (Konflikt) wird der Trip-Ausgang gesetzt.
- **DigitalOutput_Q1** – Typ: `logiBUS::io::DQ::logiBUS_QX` Parameter: `QI = TRUE`, `Output = Output_Q1` Digitalausgang für das Auf-Signal.
- **DigitalOutput_Q2** – Typ: `logiBUS::io::DQ::logiBUS_QX` Parameter: `QI = TRUE`, `Output = Output_Q2` Digitalausgang für das Ab-Signal.
- **Trip_Anzeige** – Typ: `logiBUS::io::DQ::logiBUS_QX` Parameter: `QI = TRUE`, `Output = Output_Q4` Digitalausgang zur Anzeige des Trip-Zustands.

Programmablauf und Verbindungen

Die Übung ist als SubAppType angelegt, in der die gesamte Logik abläuft. Die Verbindungen sind wie folgt:

Ereignisverbindung	Quelle	Ziel	Datenverbindung	Quelle	Ziel
IND → EI_UP	DigitalInput_I1	ILOCK	IN → DI_UP	DigitalInput_I1	ILOCK
IND → EI_DOWN	DigitalInput_I2	ILOCK	IN → DI_DOWN	DigitalInput_I2	ILOCK
IND → EI_RESET	DigitalInput_Reset	ILOCK	–	–	–
EO_UP → REQ	ILOCK	DigitalOutput_Q1	DO_UP → OUT	ILOCK	DigitalOutput_Q1
EO_DOWN → REQ	ILOCK	DigitalOutput_Q2	DO_DOWN → OUT	ILOCK	DigitalOutput_Q2
EO_TRIP → REQ	ILOCK	Trip_Anzeige	DO_TRIP → OUT	ILOCK	Trip_Anzeige

Ablauf:

1. Ein steigende Flanke auf einem der Eingänge (I1 für AUF, I2 für AB) erzeugt ein Ereignis, das den ILOCK-Baustein an seinem entsprechenden Ereigniseingang aktiviert.
2. Der ILOCK prüft, ob ein Konflikt vorliegt (beide Eingänge gleichzeitig aktiv).
   - **Kein Konflikt:** Der gewünschte Ausgang (DO_UP oder DO_DOWN) wird gesetzt und der zugehörige Ausgangstreiber (Q1 oder Q2) geschaltet.
   - **Konflikt:** Es wird kein Ausgang gesetzt, stattdessen wird der Trip-Ausgang (DO_TRIP) aktiviert und über `Trip_Anzeige` ausgegeben. Die Ausgänge Q1 und Q2 bleiben aus.
3. Ein anliegendes Rücksetzsignal (I3) kann den Trip zurücksetzen und die normale Funktion wiederherstellen. Solange der Konflikt andauert, führt ein erneuter Reset nicht zu einer Freigabe.

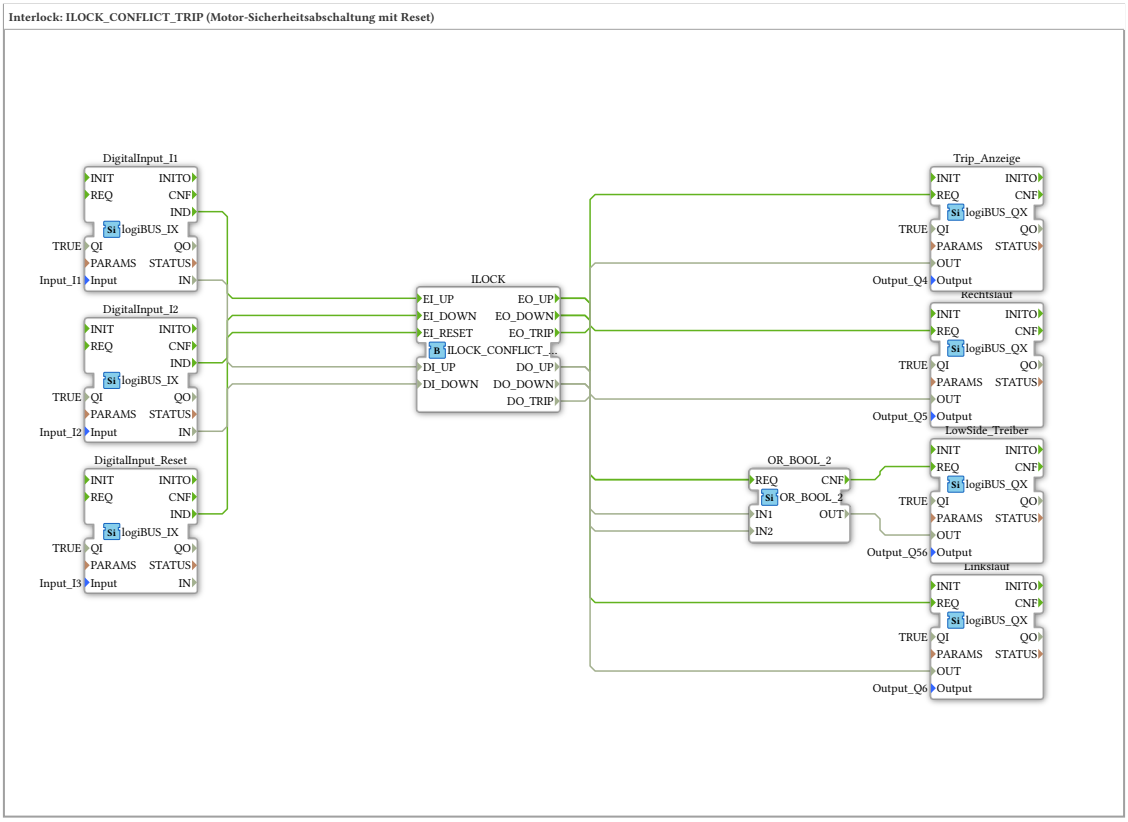
Zusammenfassung

Die Übung **Uebung_204** demonstriert die Verwendung des Interlock-Bausteins `ILOCK_CONFLICT_TRIP`. Sie zeigt, wie durch zwei gegensätzliche Stellsignale (z. B. Auf/Ab) ein Konflikt erkannt und durch einen Trip abgesichert wird. Der Baustein erfordert eine explizite Rücksetzung nach einem Konflikt. Dieses Verhalten ist typisch für Sicherheitsanwendungen in der Automatisierungstechnik. Die Übung eignet sich für Einsteiger, die grundlegende Verriegelungsmechanismen mit 4diac und logiBUS-Funktionsbausteinen erlernen möchten.

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

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

2.2.1.366 Uebung_204b: Interlock: ILOCK_CONFLICT_TRIP (Motor-Sicherheitsabschaltung mit Reset)



• – ○ • – ○ • – ○ •

Einleitung

In dieser Übung wird eine **Motor-Sicherheitsabschaltung mit Reset** realisiert. Sie basiert auf dem Funktionsbaustein `ILOCK_CONFLICT_TRIP`, der eine Verriegelung (Interlock) für zwei gegenläufige Motorrichtungen (Rechts- und Linkslauf) implementiert. Tritt ein Konflikt auf (beide Richtungen gleichzeitig aktiv), wird der Motor gestoppt und ein Alarm (Trip) ausgelöst. Ein separater Reset-Eingang erlaubt das Zurücksetzen des Trip-Zustands.

Die Steuerung erfolgt über drei digitale Eingänge:

- I1 – Anforderung Rechtslauf
- I2 – Anforderung Linkslauf
- I3 – Reset

Als Ausgänge stehen zur Verfügung:

- Q5 – Rechtslauf
- Q6 – Linkslauf
- Q4 – Trip-Anzeige
- Q56 – Low-Side-Treiber (gemeinsame Freigabe für beide Motorrichtungen)

Verwendete Funktionsbausteine (FBs)

Die Übung verwendet folgende Funktionsbausteine:

- **`DigitalInput_I1`** (Typ: `logiBUS::io::DI::logiBUS_IX`)
  - Parameter: `QI` = TRUE, `Input` = `Input_I1`
  - Wandelt das digitale Eingangssignal I1 in ein internes Signal um.
- **`DigitalInput_I2`** (Typ: `logiBUS::io::DI::logiBUS_IX`)
  - Parameter: `QI` = TRUE, `Input` = `Input_I2`
  - Wandelt das digitale Eingangssignal I2 in ein internes Signal um.

- **DigitalInput_Reset** (Typ: logiBUS::io::DI::logiBUS_IX)
  - Parameter: QI = TRUE, Input = Input_I3
  - Wandelt das digitale Eingangssignal I3 (Reset) in ein internes Signal um.
- **ILOCK** (Typ: logiBUS::signalprocessing::interlock::ILOCK_CONFLICT_TRIP)
  - Keine Parameter
  - Kernbaustein dieser Übung. Er realisiert die Verriegelungslogik mit Konflikterkennung und Trip-Funktion.
- **Rechtslauf** (Typ: logiBUS::io::DQ::logiBUS_QX)
  - Parameter: QI = TRUE, Output = Output_Q5
  - Steuert den Ausgang Q5 für Rechtslauf des Motors.
- **Linkslauf** (Typ: logiBUS::io::DQ::logiBUS_QX)
  - Parameter: QI = TRUE, Output = Output_Q6
  - Steuert den Ausgang Q6 für Linkslauf des Motors.
- **Trip_Anzeige** (Typ: logiBUS::io::DQ::logiBUS_QX)
  - Parameter: QI = TRUE, Output = Output_Q4
  - Steuert den Ausgang Q4 als Anzeige für den Trip-Zustand.
- **LowSide_Treiber** (Typ: logiBUS::io::DQ::logiBUS_QX)
  - Parameter: QI = TRUE, Output = Output_Q56
  - Steuert den Ausgang Q56 als gemeinsame Freigabe (Low-Side-Treiber) für beide Motorrichtungen.
- **OR_2_BOOL** (Typ: iec61131::bitwiseOperators::OR_2_BOOL)
  - Keine Parameter
  - Logisches ODER-Gatter; verknüpft die Signale für Rechts- und Linkslauf, um den Low-Side-Treiber anzusteuern.

Programmablauf und Verbindungen

Der Ablauf gliedert sich in folgende Schritte:

1. **Eingangserfassung**: Die drei digitalen Eingänge (I1, I2, I3) werden über die entsprechenden logiBUS_IX-Bausteine eingelesen.
   - DigitalInput_I1 liefert den Rechtslauf-Wunsch (BOOL) und ein Ereignis IND.
   - DigitalInput_I2 liefert den Linkslauf-Wunsch und ein Ereignis IND.
   - DigitalInput_Reset liefert das Reset-Signal und ein Ereignis IND.
2. **Verarbeitung im ILOCK-Baustein**:
   - Der Baustein ILOCK empfängt die Ereignisse von den Eingängen:
     - EI_UP wird durch DigitalInput_I1.IND getriggert.
     - EI_DOWN wird durch DigitalInput_I2.IND getriggert.
     - EI_RESET wird durch DigitalInput_Reset.IND getriggert.
   - Die Datenwerte (BOOL) werden über die entsprechenden Datenports übertragen:
     - DI_UP von DigitalInput_I1.IN
     - DI_DOWN von DigitalInput_I2.IN
   - Der Baustein entscheidet basierend auf seiner internen Zustandslogik, ob die Anforderung gültig ist, ein Konflikt vorliegt oder ein Reset durchgeführt wird.
3. **Ausgabe der Motorrichtungen**:
   - Bei gültiger Rechtslauf-Anforderung erzeugt ILOCK ein Ereignis EO_UP und setzt den Datenausgang DO_UP auf TRUE.
   - Bei gültiger Linkslauf-Anforderung erzeugt ILOCK ein Ereignis EO_DOWN und setzt DO_DOWN auf TRUE.
   - Im Fehlerfall (Konflikt) erzeugt ILOCK ein Ereignis EO_TRIP und setzt DO_TRIP auf TRUE.
   - Die Ereignisse werden an die entsprechenden Ausgangsbausteine weitergeleitet:
     - EO_UP ⟶ Rechtslauf.REQ
     - EO_DOWN ⟶ Linkslauf.REQ
     - EO_TRIP ⟶ Trip_Anzeige.REQ
   - Die Datenwerte werden über die Datenverbindungen auf die Ausgangsbausteine übertragen:

  - `DO_UP` ⟶ `Rechtslauf.OUT`
  - `DO_DOWN` ⟶ `Linkslauf.OUT`
  - `DO_TRIP` ⟶ `Trip_Anzeige.OUT`

4. **Low-Side-Treiber**:
   - Der Low-Side-Treiber (Ausgang Q56) wird aktiviert, sobald entweder Rechts- oder Linkslauf aktiv ist.
   - Dazu werden die Ereignisse `EO_UP` und `EO_DOWN` (beide) an den Baustein `OR_2_BOOL.REQ` geleitet.
   - Die Datenwerte `DO_UP` und `DO_DOWN` werden an die Eingänge `IN1` bzw. `IN2` des ODER-Gatters geführt.
   - Der Ausgang `OR_2_BOOL.OUT` ist TRUE, wenn mindestens eine der beiden Anforderungen aktiv ist.
   - Das Ereignis `OR_2_BOOL.CNF` triggert den Baustein `LowSide_Treiber.REQ`, und der Datenwert `OR_2_BOOL.OUT` wird an `LowSide_Treiber.OUT` übergeben.

Lernziele

- Verständnis des Interlock-Konzepts für Motorsteuerungen
- Umgang mit dem Baustein `ILOCK_CONFLICT_TRIP` (Konflikt-/Trip-Logik)
- Verknüpfung von Ereignis- und Datenflüssen in der 4diac-IDE
- Anwendung eines ODER-Gatters zur gemeinsamen Freigabe
- Fehlerbehandlung durch Reset-Mechanismus

Schwierigkeitsgrad

Fortgeschritten – Grundkenntnisse in der 4diac-IDE und im Umgang mit Funktionsbausteinen werden vorausgesetzt.

Vorkenntnisse

- Grundlagen der IEC 61499
- Ablaufsteuerungen und Verriegelungen
- Ein-/Ausgabe-Konfiguration mit logiBUS-Bausteinen

Start der Übung

1. Öffnen Sie die 4diac-IDE und laden Sie die Übung `Uebung_204b`.
2. Stellen Sie sicher, dass die benötigten logiBUS-Bibliotheken importiert sind (siehe CompilerInfo).
3. Überprüfen Sie die Verbindungen zwischen den Bausteinen.
4. Simulieren Sie das Verhalten durch Anlegen der Eingangssignale I1, I2 und I3.

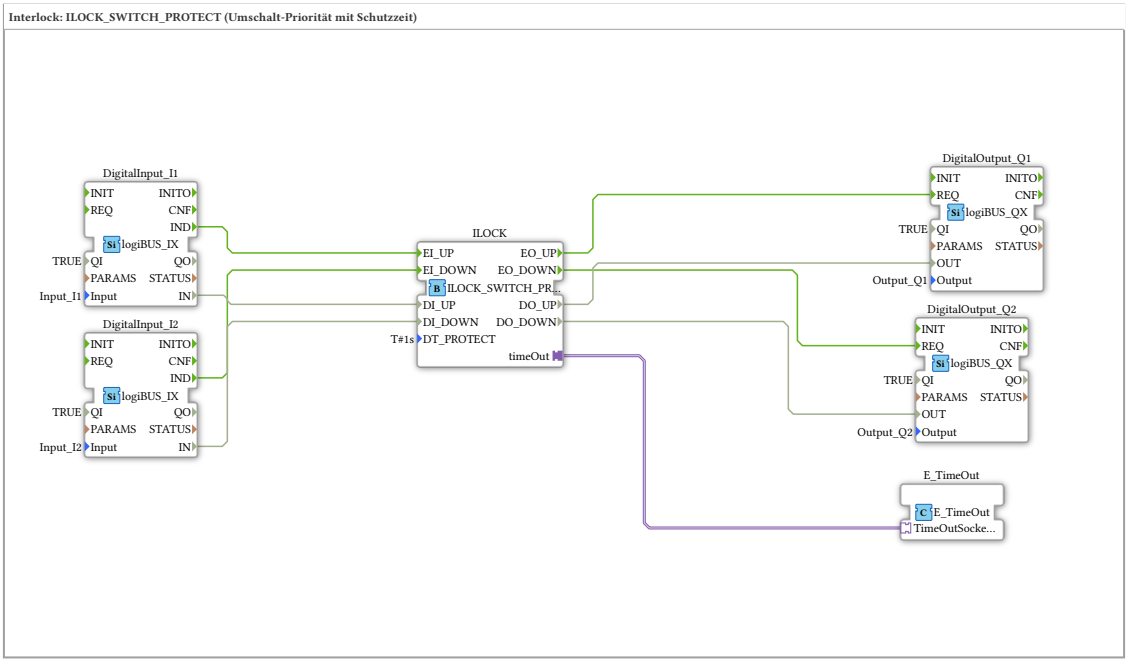
Zusammenfassung

Die Übung `Uebung_204b` demonstriert den Einsatz des Funktionsbausteins `ILOCK_CONFLICT_TRIP` für eine Motor-Sicherheitsabschaltung. Durch die Kombination von drei digitalen Eingängen (zwei Richtungswünsche und ein Reset) wird eine Verriegelung realisiert, die Konflikte erkennt und im Fehlerfall einen Trip auslöst. Die Ansteuerung der Ausgänge erfolgt über getrennte Kanäle für Rechtslauf, Linkslauf sowie eine gemeinsame Low-Side-Freigabe. Die Lösung zeigt exemplarisch, wie sicherheitsgerichtete Steuerungen mit der 4diac-IDE umgesetzt werden können.

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

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

2.2.1.367 Uebung_205: Interlock: ILOCK_SWITCH_PROTECT (Umschalt-Priorität mit Schutzzeit)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert den Einsatz des Funktionsbausteins `ILOCK_SWITCH_PROTECT` zur Realisierung einer Umschalt-Priorität mit Schutzzeit (Interlock). Zwei digitale Eingänge (I1, I2) steuern zwei digitale Ausgänge (Q1, Q2) über eine verriegelte Logik. Der `ILOCK_SWITCH_PROTECT` stellt sicher, dass nach einer Umschaltung eine konfigurierbare Schutzzeit (`DT_PROTECT`) eingehalten wird, bevor erneut umgeschaltet werden kann. Dies verhindert ein schnelles, unerwünschtes Hin- und Herschalten (Pendeln). Die Ausgänge werden über ereignisgesteuerte Ausgangsbausteine angesteuert. Ein `E_TimeOut`-Baustein ist über einen Adapter an den `ILOCK` angeschlossen und ermöglicht eine Zeitüberwachung der Schutzzeit.

Verwendete Funktionsbausteine (FBs)

Bausteine der SubApplikation `Uebung_205`

Bausteinname	Typ	Parameter	Beschreibung
`DigitalInput_I1`	`logiBUS::io::DI::logiBUS_IX`	`QI = TRUE, Input = Input_I1`	Liest den ersten digitalen Eingang (Hardwareadresse `Input_I1`). Das Ereignis `IND` signalisiert eine Wertänderung.
`DigitalInput_I2`	`logiBUS::io::DI::logiBUS_IX`	`QI = TRUE, Input = Input_I2`	Liest den zweiten digitalen Eingang. Das Ereignis `IND` signalisiert eine Wertänderung.
`logiBUS_ILOCK`	`signalprocessing::interlock::ILOCK_SWITCH_PROTECT`	`DT_PROTECT=T#1s`	Kernbaustein: Realisiert die Umschalt-Priorität mit Schutzzeit von 1 Sekunde.
`DigitalOutput_Q1`	`logiBUS::io::DQ::logiBUS_QX`	`QI = TRUE, Output = Output_Q1`	Setzt den ersten digitalen Ausgang (Hardwareadresse `Output_Q1`) auf

			den Wert des Dateneingangs `OUT`, wenn das Ereignis `REQ` eintrifft.
`DigitalOutput_Q2`	`logiBUS::io::DQ::logiBUS_QX`	`QI = TRUE, Output = Output_Q2`	Setzt den zweiten digitalen Ausgang (Hardwareadresse `Output_Q2`) auf den Wert des Dateneingangs `OUT`, wenn das Ereignis `REQ` eintrifft.
`E_TimeOut`	`iec61499::events::E_TimeOut`	–	Zeitgeber-Baustein, der über den Adapter `timeOut` mit dem `ILOCK` verbunden ist. Er überwacht die Einhaltung der Schutzzeit.

Programmablauf und Verbindungen

1. **Ereignissteuerung:**
   - Eine steigende/fallende Flanke an `DigitalInput_I1` löst das Ereignis `DigitalInput_I1.IND` aus → verbunden mit `ILOCK.EI_UP`.
   - Eine Flanke an `DigitalInput_I2` löst `DigitalInput_I2.IND` aus → verbunden mit `ILOCK.EI_DOWN`.
   - Der `ILOCK` generiert bei erfolgreicher Umschaltung das Ereignis `EO_UP` (für den oberen Ausgang) bzw. `EO_DOWN` (für den unteren Ausgang).
   - `ILOCK.EO_UP` ist mit `DigitalOutput_Q1.REQ` verbunden (Ausgang Q1 schaltet).
   - `ILOCK.EO_DOWN` ist mit `DigitalOutput_Q2.REQ` verbunden (Ausgang Q2 schaltet).
2. **Datenpfad:**
   - Der Wert des Eingangs `DigitalInput_I1.IN` wird auf `ILOCK.DI_UP` übertragen.
   - Der Wert von `DigitalInput_I2.IN` wird auf `ILOCK.DI_DOWN` übertragen.
   - Der ILOCK gibt über `DO_UP` den Zustand für den oberen Ausgang (1 = aktiv) an `DigitalOutput_Q1.OUT` weiter.
   - Entsprechend gibt `DO_DOWN` den Zustand an `DigitalOutput_Q2.OUT` weiter.
3. **Schutzzeit-Überwachung:**
   - Der `ILOCK_SWITCH_PROTECT` verfügt über einen Adapteranschluss `timeOut`, der mit dem `E_TimeOut.TimeOutSocket` verbunden ist. Dieser Adapter erlaubt es dem `E_TimeOut`, die Schutzzeit (hier 1 Sekunde) zu überwachen und bei Ablauf ein Ereignis zu erzeugen, falls nötig.
4. **Funktionsweise des Interlocks:**
   - Der `ILOCK_SWITCH_PROTECT` arbeitet mit Priorität: Welcher Eingang zuerst aktiv wird, setzt den entsprechenden Ausgang. Solange die Schutzzeit (`DT_PROTECT = 1s`) läuft, wird der andere Eingang ignoriert. Erst nach Ablauf der Schutzzeit kann wieder umgeschaltet werden.
   - Dies verhindert ein schnelles Hin- und Herschalten zwischen den beiden Ausgängen (z.B. bei mechanischen Prellern oder schnellen Tastern).

Lernziele:

- Verwendung des Interlock-Bausteins `ILOCK_SWITCH_PROTECT` mit Schutzzeit.
- Verständnis für ereignisgesteuerte Kommunikation (IND → EI, EO → REQ).
- Einbindung eines Zeitgeber-Adapters zur Überwachung der Schutzzeit.
- Konfiguration von Hardware-Ein-/Ausgängen (`logiBUS_IX`, `logiBUS_QX`).

Schwierigkeitsgrad: Mittel **Vorkenntnisse:** Grundlagen der Ereignissteuerung in 4diac, Umgang mit Ein-/Ausgangsbausteinen.

Zusammenfassung

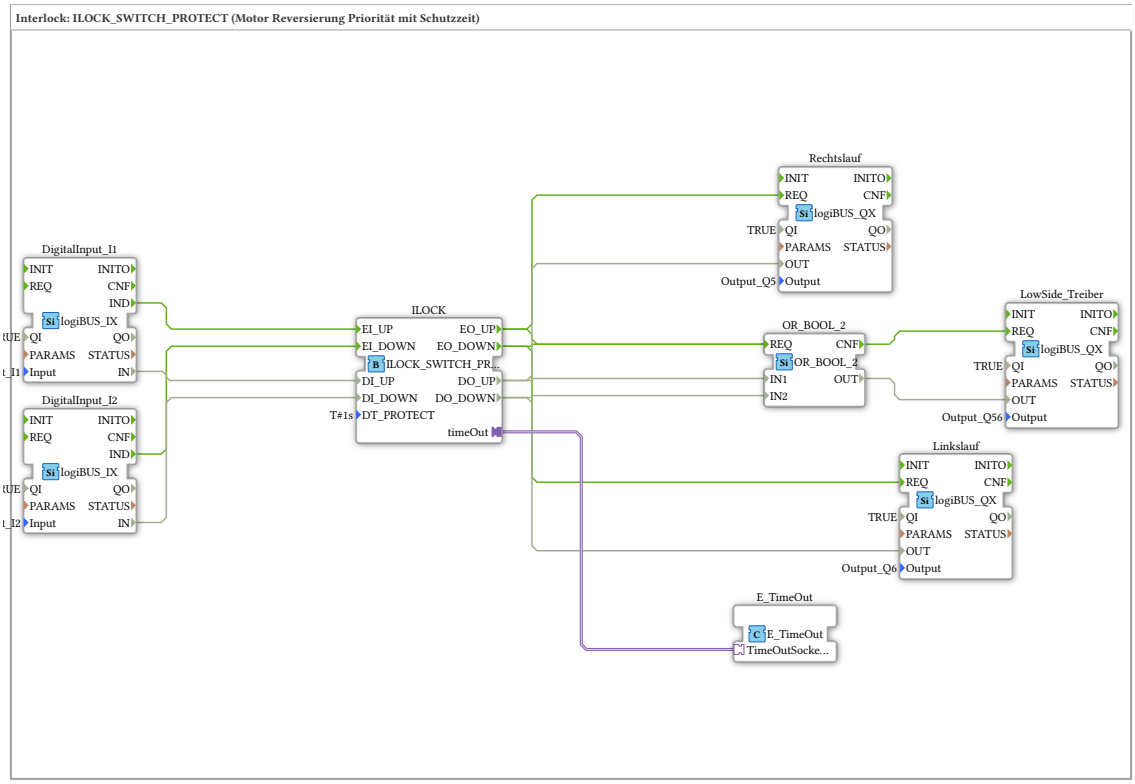
Die Übung **Uebung_205** zeigt die praktische Anwendung eines Interlocks mit Schutzzeit mithilfe des `ILOCK_SWITCH_PROTECT`-Bausteins. Zwei digitale Eingänge steuern zwei Ausgänge, wobei eine Umschaltung nur nach Ablauf einer konfigurierbaren Schutzzeit möglich ist. Der Aufbau verdeutlicht die

Trennung von Ereignis- und Datenpfaden sowie die Integration von Zeitüberwachung über Adapter. Diese Schaltung eignet sich typischerweise für Anwendungen, bei denen zwei Aktoren nicht gleichzeitig aktiv sein dürfen und ein schnelles Umschalten vermieden werden muss (z.B. Motor-Richtungsumkehr, Ventilsteuerung).

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

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

2.2.1.368 Uebung_205b: Interlock: ILOCK_SWITCH_PROTECT (Motor Reversierung Priorität mit Schutzzeit)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung zeigt die Realisierung einer **Motor-Umkehrverriegelung** mit Prioritätsschutz und einer Schutzzeit. Mit Hilfe des Funktionsbausteins `ILOCK_SWITCH_PROTECT` wird sichergestellt, dass ein Motor nicht gleichzeitig in beide Drehrichtungen (Rechts- und Linkslauf) geschaltet werden kann. Ein zusätzlicher Low-Side-Treiber schaltet die gemeinsame Versorgung. Die Schutzzeit `DT_PROTECT` von 1 s verhindert ein zu schnelles Umschalten und schützt die Leistungselektronik.

Verwendete Funktionsbausteine (FBs)

Eingangsbausteine – Digitale Eingänge

- **DigitalInput_I1** (Typ `logiBUS::io::DI::logiBUS_IX`)
  - Parameter: `QI = TRUE`, `Input = Input_I1`
  - Funktion: Erfasst das Signal des ersten Tasters/Sensors (z. B. „Rechts starten“) und gibt es als Ereignis `IND` und Datenwert `IN` weiter.
- **DigitalInput_I2** (Typ `logiBUS::io::DI::logiBUS_IX`)
  - Parameter: `QI = TRUE`, `Input = Input_I2`
  - Funktion: Erfasst das Signal des zweiten Tasters/Sensors (z. B. „Links starten“) und gibt es als Ereignis `IND` und Datenwert `IN` weiter.

Verriegelungsbaustein

- **ILOCK** (Typ `logiBUS::signalprocessing::interlock::ILOCK_SWITCH_PROTECT`)
  - Parameter: `DT_PROTECT = T#1s`
  - Funktion: Realisiert die Verriegelung zwischen zwei Richtungen.
    - Ereigniseingänge: `EI_UP` (Rechts anfordern), `EI_DOWN` (Links anfordern)
    - Datenausgänge: `DO_UP`, `DO_DOWN` (Freigabe der Richtungen)
    - Ereignisausgänge: `EO_UP`, `EO_DOWN` (Bestätigung der Freigabe)
    - Intern wird bei einer Richtungsänderung die Schutzzeit `DT_PROTECT` gestartet, bevor die andere Richtung freigegeben werden kann.
    - Adapter `timeOut` gibt den Timerstatus an den `E_TimeOut`-Baustein weiter.

Ausgangsbausteine

- **Rechtslauf** (Typ `logiBUS::io::DQ::logiBUS_QX`)
  - Parameter: `QI = TRUE`, `Output = Output_Q5`
  - Funktion: Schaltet den Ausgang für die Rechtslaufspule (Q5), wenn über `REQ` und `IN` die Freigabe vom ILOCK kommt.
- **Linkslauf** (Typ `logiBUS::io::DQ::logiBUS_QX`)
  - Parameter: `QI = TRUE`, `Output = Output_Q6`
  - Funktion: Schaltet den Ausgang für die Linkslaufspule (Q6), wenn über `REQ` und `IN` die Freigabe vom ILOCK kommt.
- **LowSide_Treiber** (Typ `logiBUS::io::DQ::logiBUS_QX`)
  - Parameter: `QI = TRUE`, `Output = Output_Q56`
  - Funktion: Schaltet die gemeinsame Versorgung (Q56) für beide Richtungen. Wird aktiviert, sobald eine der beiden Freigaben (`DO_UP` oder `DO_DOWN`) anliegt.

Logikbaustein

- **OR_2_BOOL** (Typ `iec61131::bitwiseOperators::OR_2_BOOL`)
  - Funktion: Logische ODER-Verknüpfung der beiden Freigabesignale (`DO_UP` und `DO_DOWN`).
    - Eingänge: `IN1`, `IN2`
    - Ausgang: `OUT` – ergibt `TRUE`, wenn mindestens eine Richtung freigegeben ist.

Timerbaustein

- **E_TimeOut** (Typ `iec61499::events::E_TimeOut`)
  - Funktion: Dient der Visualisierung/Überwachung der Schutzzeit. Er ist über den Adapter `timeOut` mit dem ILOCK verbunden und zeigt den Status des laufenden Timers an (z. B. für Diagnosezwecke).

Programmablauf und Verbindungen

1. **Eingangsverarbeitung**
   - Die digitalen Eingänge `Input_I1` (Rechts) und `Input_I2` (Links) werden über die Bausteine `DigitalInput_I1` und `DigitalInput_I2` erfasst.
   - Bei einer Flanke wird ein Ereignis (`IND`) an den ILOCK gesendet.
2. **Verriegelung durch ILOCK**
   - Der ILOCK prüft, ob eine Richtungsänderung zulässig ist.
   - Bei einem Umschaltwunsch wird die Schutzzeit von 1 s gestartet.
   - Erst nach Ablauf der Schutzzeit wird die neue Richtung freigegeben und das entsprechende Ereignis (`EO_UP`/`EO_DOWN`) sowie der Datenwert (`DO_UP`/`DO_DOWN`) ausgegeben.
   - Der Adapter `timeOut` liefert den Timerstatus an den `E_TimeOut`, der z. B. in einer HMI-Anzeige verwendet werden kann.
3. **Ansteuerung der Ausgänge**
   - `EO_UP` triggert den Baustein `Rechtslauf`, der den Ausgang `Output_Q5` setzt.
   - `EO_DOWN` triggert den Baustein `Linkslauf`, der den Ausgang `Output_Q6` setzt.
   - Parallel werden die Datenausgänge `DO_UP` und `DO_DOWN` an die ODER-Verknüpfung (`OR_2_BOOL`) weitergeleitet.
   - Das ODER-Signal aktiviert den `LowSide_Treiber`, der die gemeinsame Versorgung `Output_Q56` einschaltet. Dadurch fließt nur dann Strom, wenn mindestens eine Richtung aktiv ist.

4. **Schutzmechanismus**
   - Die Schutzzeit verhindert, dass bei schnellem Wechsel der Anforderung (Prellen, Fehlbedienung) die Ausgänge zu schnell umschalten, und schützt so die Motorbrücke.

Zusammenfassung

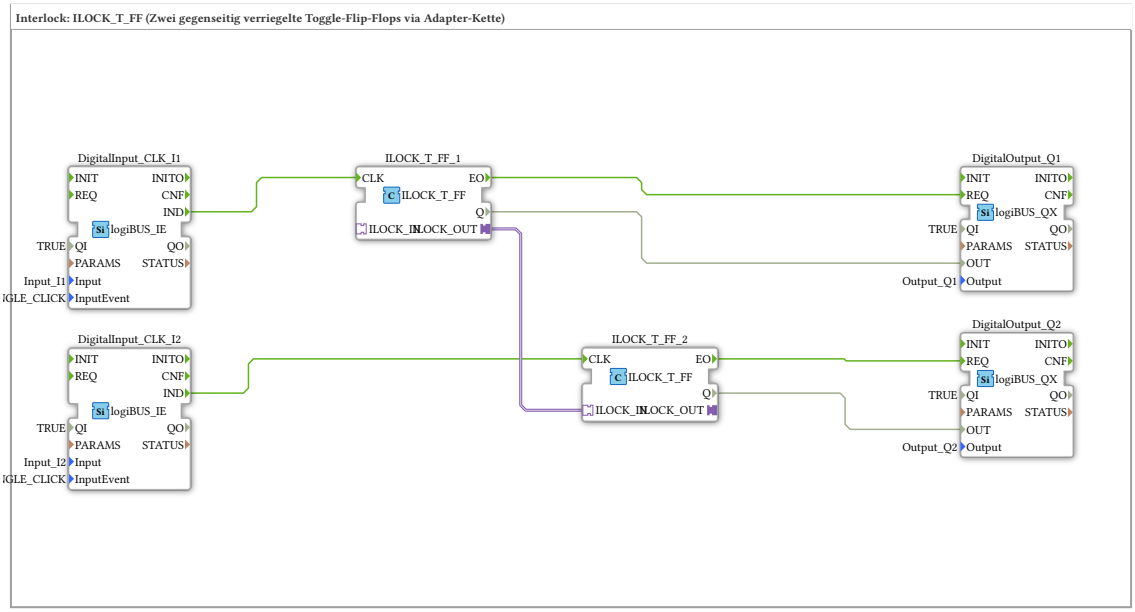
Die Übung 205b demonstriert eine industriegerechte Motor-Umkehrschaltung mit **Prioritätsverriegelung** (`ILOCK_SWITCH_PROTECT`). Durch die integrierte **Schutzzeit** wird ein zu schnelles Umschalten zwischen Rechts- und Linkslauf unterbunden. Ein Low-Side-Treiber sorgt für die saubere Trennung der Versorgung. Der Einsatz von logiBUS- und IEC-61131-Bausteinen macht die Lösung hardwarenah und in SPS-Projekten wiederverwendbar.

Lernziele:

- Verriegelungskonzepte für Motorsteuerungen verstehen
- Umgang mit dem Funktionsbaustein `ILOCK_SWITCH_PROTECT`
- Anwendung von ODER-Verknüpfungen zur Freigabe von Gemeinschaftsausgängen
- Interpretation von Schutzzeiten in der Steuerungstechnik

Schwierigkeitsgrad: Fortgeschrittene Einsteiger (Grundlagen der IEC 61499 und logiBUS-Hardware werden vorausgesetzt).

2.2.1.369 Uebung_206: Interlock: ILOCK_T_FF (Zwei gegenseitig verriegelte Toggle-Flip-Flops via Adapter-Kette)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert die gegenseitige Verriegelung (Interlock) zweier Toggle-Flip-Flops. Jeder Taster (I1 und I2) steuert einen separaten Ausgang (Q1 bzw. Q2). Die Besonderheit besteht darin, dass die beiden Flip-Flops über eine Adapterverbindung („ILOCK“) miteinander verbunden sind, sodass immer nur einer der beiden Ausgänge aktiv sein kann. Wird der andere Taster betätigt, wird der zuvor aktive Ausgang zurückgesetzt und der neue Ausgang gesetzt. Es entsteht eine einfache Wechselblinkschaltung mit gegenseitiger Verriegelung.

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_CLK_I1** und **DigitalInput_CLK_I2**: `logiBUS::io::DI::logiBUS_IE`
  - Parameter: `QI = TRUE`, `Input = Input_I1` bzw. `Input_I2`, `InputEvent = BUTTON_SINGLE_CLICK`
  - Diese Bausteine wandeln das vom Hardware-Eingang (z. B. Taster) kommende Signal in ein Ereignis (`IND`) um, das als Takt für die Flip-Flops dient.
- **ILOCK_T_FF_1** und **ILOCK_T_FF_2**: `logiBUS::signalprocessing::interlock::ILOCK_T_FF`

  - Diese Bausteine sind spezielle Toggle-Flip-Flops mit einer Interlock-Schnittstelle (Adapter). Bei jedem Takt (Ereignis an `CLK`) toggelt der Ausgang `Q`. Zusätzlich kann der Ausgangszustand über den Adapter (`ILOCK_IN`) von außen zurückgesetzt werden.
- **DigitalOutput_Q1** und **DigitalOutput_Q2**: `logiBUS::io::DQ::logiBUS_QX`
  - Parameter: `QI = TRUE`, `Output = Output_Q1` bzw. `Output_Q2`
  - Diese Bausteine geben den Datenwert an den Hardware-Ausgang weiter, sobald ein Ereignis an `REQ` anliegt.

Programmablauf und Verbindungen

1. **Eingangsverarbeitung**
   Jeder Taster (I1, I2) wird über einen `logiBUS_IE`-Baustein eingelesen. Bei jedem Single-Click erzeugt der Baustein ein Ereignis am Ausgang `IND`.
2. **Toggle-Flip-Flops**
   Das Ereignis von `DigitalInput_CLK_I1` wird mit dem Eingang `CLK` von `ILOCK_T_FF_1` verbunden. Analog wird `DigitalInput_CLK_I2` mit `ILOCK_T_FF_2` verbunden. Bei jedem Ereignis wechselt der Ausgang `Q` des entsprechenden Flip-Flops seinen Zustand (Toggle).
3. **Gegenseitige Verriegelung (Interlock)**
   Der Adapterausgang `ILOCK_OUT` von `ILOCK_T_FF_1` ist mit dem Adaptereingang `ILOCK_IN` von `ILOCK_T_FF_2` verbunden (bidirektionale Kette). Dadurch wird aktiviert, sobald `ILOCK_T_FF_1` gesetzt ist, der andere Baustein zurückgesetzt. Gleichzeitig wird über die Verbindung auch der umgekehrte Fall realisiert: Setzt `ILOCK_T_FF_2`, wird `ILOCK_T_FF_1` zurückgesetzt. Somit kann immer nur einer der beiden Ausgänge den Wert `TRUE` (also „aktiv“) haben.
4. **Ausgangsansteuerung**
   Die Ausgangsereignisse `EO` der Flip-Flops triggern die `REQ`-Eingänge der Ausgangsbausteine. Gleichzeitig werden die Datenwerte `Q` an die `OUT`-Eingänge der Ausgangsbausteine weitergeleitet. Damit werden die Hardware-Ausgänge Q1 und Q2 entsprechend dem Zustand der Flip-Flops gesetzt.

Lernziele dieser Übung:

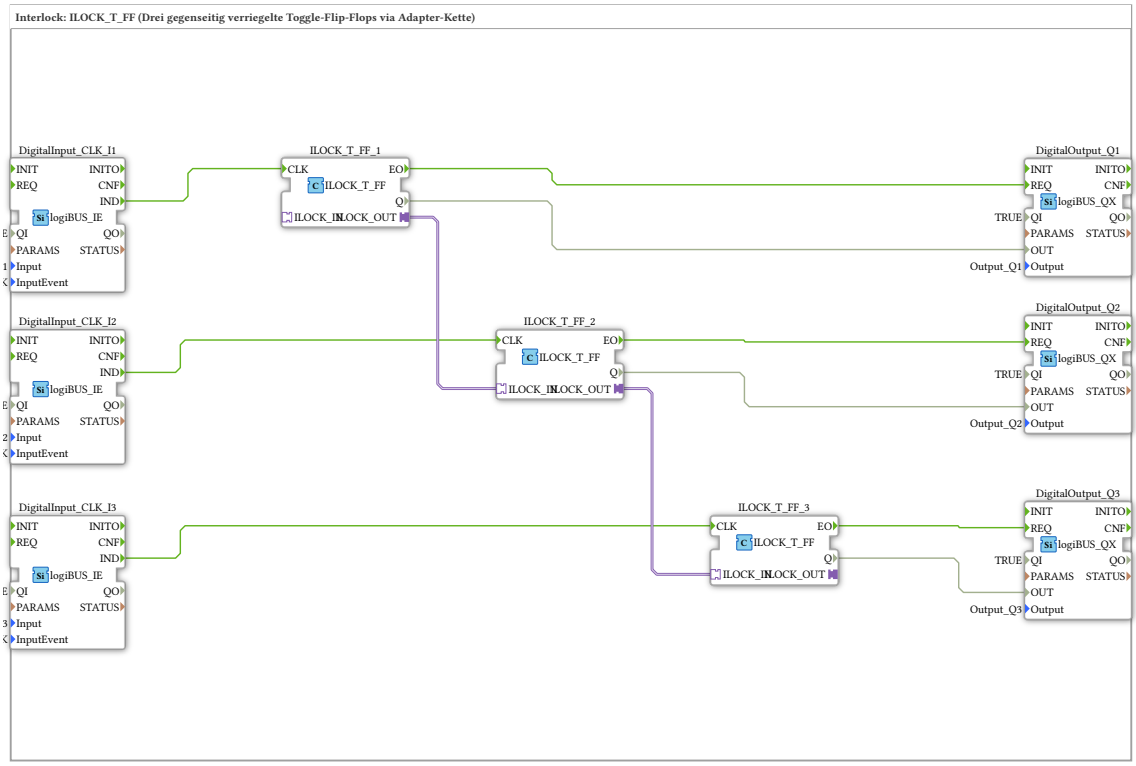
- Verständnis der Funktionsweise eines Toggle-Flip-Flops (Toggle bei jedem Takt).
- Anwendung einer Adapterverbindung zur Realisierung einer gegenseitigen Verriegelung (Interlock).
- Einlesen von Tastern mit Single-Click-Ereignissen und Ausgabe an digitale Ausgänge.
- Aufbau einer einfachen Zustandssteuerung mit zwei sich ausschließenden Zuständen.

Schwierigkeitsgrad: Mittel
Vorkenntnisse: Grundlagen der IEC 61499, Ereignis- und Datenverbindungen, Umgang mit Adaptern.

Zusammenfassung

Die Übung 206 zeigt eine elegante Lösung für eine gegenseitig verriegelte Schaltung mit zwei Toggle-Flip-Flops. Durch die Verwendung des vorgefertigten Interlock-Bausteins `ILOCK_T_FF` und der Adapterverbindung zwischen beiden Instanzen wird sichergestellt, dass stets nur ein Ausgang aktiv ist. Dieses Prinzip findet sich häufig in Steuerungen, bei denen mehrere Aktoren nicht gleichzeitig aktiv sein dürfen (z. B. Richtungswechsel eines Motors, Schrankensteuerung).

2.2.1.370 Uebung_206b: Interlock: ILOCK_T_FF (Drei gegenseitig verriegelte Toggle-Flip-Flops via Adapter-Kette)



• – ○ • – ○ • – ○ •

Einleitung

In dieser Übung wird eine Anwendung mit drei gegenseitig verriegelten Toggle-Flip-Flops realisiert. Drei Taster (Digitaleingänge) steuern jeweils ein ILOCK_T_FF, welches seinen Ausgang bei jedem Tastendruck umschaltet. Die drei Flip-Flops sind über eine Adapter-Kette bidirektional miteinander verbunden, sodass immer nur ein Ausgang aktiv sein kann (Interlock). Die Ausgänge werden auf drei digitale Ausgänge (z. B. LEDs) geführt.

Verwendete Funktionsbausteine (FBs)

Digitaleingang: logiBUS_IE

- **Typ**: logiBUS::io::DI::logiBUS_IE
- **Verwendete Instanzen**: DigitalInput_CLK_I1, DigitalInput_CLK_I2, DigitalInput_CLK_I3
- **Parameter**:
  - QI = TRUE
  - Input = Input_I1 / Input_I2 / Input_I3
  - InputEvent = BUTTON_SINGLE_CLICK
- **Funktionsweise**: Jeder Baustein erfasst einen Taster (Button) und erzeugt bei einem einzelnen Klick ein Ereignis (IND) an seinem Ereignisausgang.

Interlock-Toggle-Flip-Flop: ILOCK_T_FF

- **Typ**: logiBUS::signalprocessing::interlock::ILOCK_T_FF
- **Verwendete Instanzen**: ILOCK_T_FF_1, ILOCK_T_FF_2, ILOCK_T_FF_3
- **Parameter**: keine spezifischen Parameter im XML
- **Funktionsweise**: Ein Toggle-Flip-Flop, das bei jedem Ereignis am Eingang **CLK** seinen Ausgang **Q** umschaltet. Es besitzt zwei Adapter-Schnittstellen: **ILOCK_IN** und **ILOCK_OUT**. Über diese Adapter können mehrere ILOCK_T_FF zu einer Kette verbunden werden, wodurch sichergestellt wird, dass immer nur ein Flip-Flop in der Kette seinen Ausgang auf TRUE setzen kann (gegenseitige Verriegelung). Bei Aktivierung eines anderen Flip-Flops wird das zuvor aktive automatisch zurückgesetzt.

Digitalausgang: logiBUS_QX

- **Typ**: logiBUS::io::DQ::logiBUS_QX
- **Verwendete Instanzen**: DigitalOutput_Q1, DigitalOutput_Q2, DigitalOutput_Q3
- **Parameter**:
  - QI = TRUE
  - Output = Output_Q1 / Output_Q2 / Output_Q3
- **Funktionsweise**: Der Baustein setzt einen digitalen Ausgang (z. B. eine LED) auf den Wert, der am Dateneingang **OUT** anliegt. Die Ausgabe erfolgt bei einem Ereignis am **REQ**-Eingang.

Programmablauf und Verbindungen

1. **Eingangssignale**: Drei Taster sind an die logiBUS-Eingänge *Input_I1*, *Input_I2* und *Input_I3* angeschlossen. Jeder Tastendruck (Single Click) erzeugt ein Ereignis am zugehörigen logiBUS_IE, das über den Ereignisausgang **IND** an den **CLK**-Eingang des entsprechenden ILOCK_T_FF weitergeleitet wird.
2. **Verriegelung (Interlock)**: Die drei ILOCK_T_FF sind über ihre Adapter-Schnittstellen verbunden:
   - ILOCK_T_FF_1.ILOCK_OUT → ILOCK_T_FF_2.ILOCK_IN
   - ILOCK_T_FF_2.ILOCK_OUT → ILOCK_T_FF_3.ILOCK_IN Diese Kette bewirkt, dass nur eines der drei Flip-Flops seinen Ausgang **Q** auf TRUE setzen kann. Sobald ein anderes Flip-Flop seinen Zustand ändert, wird das vorher aktive zurückgesetzt.
3. **Ausgangssignale**: Die Ausgänge **Q** der Flip-Flops sind mit den Dateneingängen **OUT** der Digitalausgangsbausteine verbunden. Das Ereignis **EO** jedes ILOCK_T_FF (wird bei Zustandsänderung ausgelöst) triggert den **REQ**-Eingang des zugehörigen logiBUS_QX, sodass die Ausgabe aktualisiert wird.

Verbindungen im Überblick (Ereignisse & Daten):

Quelle	Ziel	Art
DigitalInput_CLK_I1.IND	ILOCK_T_FF_1.CLK	Ereignis
DigitalInput_CLK_I2.IND	ILOCK_T_FF_2.CLK	Ereignis
DigitalInput_CLK_I3.IND	ILOCK_T_FF_3.CLK	Ereignis
ILOCK_T_FF_1.EO	DigitalOutput_Q1.REQ	Ereignis
ILOCK_T_FF_2.EO	DigitalOutput_Q2.REQ	Ereignis
ILOCK_T_FF_3.EO	DigitalOutput_Q3.REQ	Ereignis
ILOCK_T_FF_1.Q	DigitalOutput_Q1.OUT	Daten
ILOCK_T_FF_2.Q	DigitalOutput_Q2.OUT	Daten
ILOCK_T_FF_3.Q	DigitalOutput_Q3.OUT	Daten

Adapterverbindungen (bidirektional):

Quelle	Ziel
ILOCK_T_FF_1.ILOCK_OUT	ILOCK_T_FF_2.ILOCK_IN
ILOCK_T_FF_2.ILOCK_OUT	ILOCK_T_FF_3.ILOCK_IN

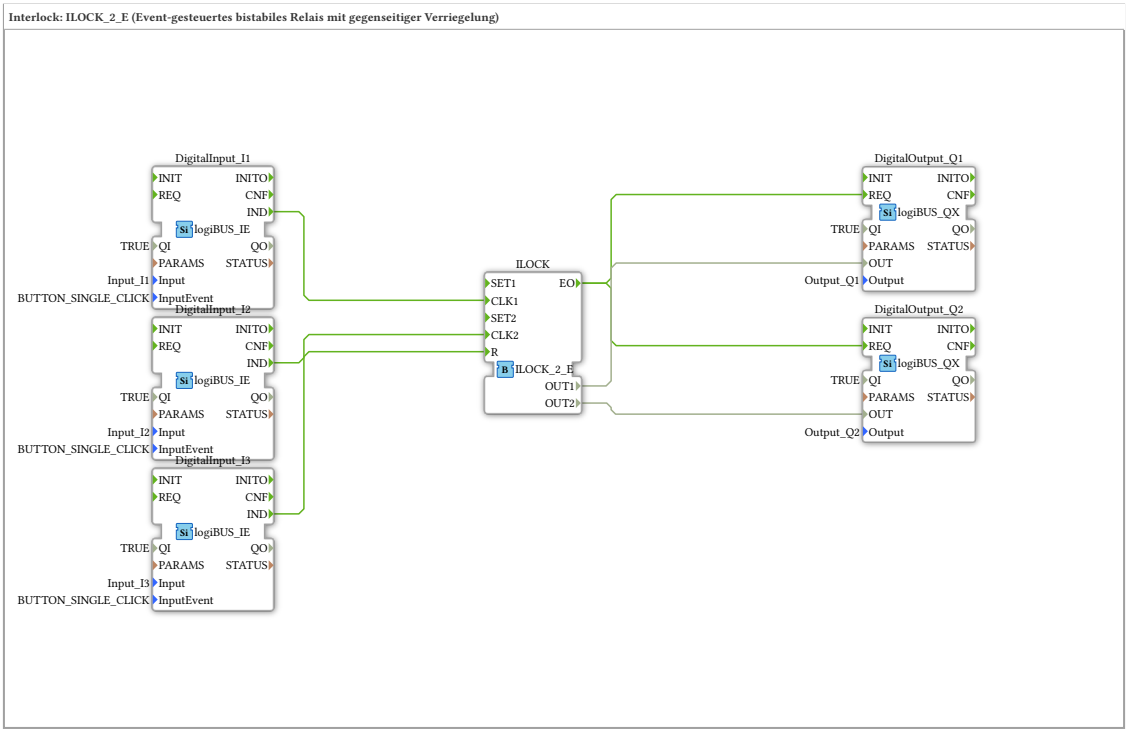
Zusammenfassung

Diese Übung demonstriert die Verwendung des ILOCK_T_FF-Bausteins zur Realisierung einer gegenseitigen Verriegelung (Interlock) von drei Toggle-Flip-Flops. Durch die Adapter-Kette wird sichergestellt, dass stets nur ein Ausgang aktiv ist, was typischerweise für Anwendungen mit wechselnden Betriebsmodi oder exklusiven Zuständen benötigt wird. Die Ein-/Ausgabe erfolgt über die logiBUS-Hardware. **Lernziele**: Verständnis von Interlock-Mechanismen, Umgang mit Adapter-Schnittstellen und Ereignisgesteuerter Logik in 4diac. **Voraussetzungen**: Grundkenntnisse der 4diac-IDE und der logiBUS-Bibliothek.

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

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

2.2.1.371 Uebung_207: Interlock: ILOCK_2_E (Event-gesteuertes bistabiles Relais mit gegenseitiger Verriegelung)



• – ○ • – ○ • – ○ •

Einleitung

Die Übung 207 realisiert ein **event-gesteuertes bistabiles Relais mit gegenseitiger Verriegelung** (Interlock). Über zwei Taster (Eingänge I1 und I2) können zwei Ausgänge (Q1 und Q2) wechselseitig gesetzt werden, wobei sich die Ausgänge gegenseitig ausschließen. Ein dritter Taster (Eingang I3) dient als Reset, um beide Ausgänge zurückzusetzen.

Diese Schaltung ist typisch für Sicherheitsanwendungen, bei denen nie beide Ausgänge gleichzeitig aktiv sein dürfen (z. B. Verriegelung von Antrieben oder Weichen).

Verwendete Funktionsbausteine (FBs)

Baustein	Typ	Kurzbeschreibung
`DigitalInput_I1`, `DigitalInput_I2`, `DigitalInput_I3`	`logiBUS_IE` (logiBUS Digital Input Event)	Wandelt einen Tastendruck (Single Click) in ein Ereignis `IND` um. Der Parameter `Input` legt den physikalischen Eingang fest (z. B. `Input_I1`).
`ILOCK`	`ILOCK_2_E` (logiBUS Interlock, eventgesteuert)	Bistabiles Relais mit zwei Ausgängen `OUT1`/`OUT2`. Die Ereignisse `CLK1` und `CLK2` setzen den jeweiligen Ausgang (mit gegenseitiger Verriegelung), `R` setzt beide zurück.
`DigitalOutput_Q1`, `DigitalOutput_Q2`	`logiBUS_QX` (logiBUS Digital Output)	Nimmt über das Ereignis `REQ` einen Datenwert (`OUT`) entgegen und gibt ihn am physikalischen Ausgang aus.

Programmablauf und Verbindungen

Die Verschaltung erfolgt innerhalb einer Subapplikation (`SubAppType`). Die nachfolgende Grafik zeigt die logische Verbindung der Funktionsbausteine:

```
[I1] → DigitalInput_I1.IND → ILOCK.CLK1
[I2] → DigitalInput_I2.IND → ILOCK.CLK2
[I3] → DigitalInput_I3.IND → ILOCK.R

ILOCK.EO → DigitalOutput_Q1.REQ
         → DigitalOutput_Q2.REQ

ILOCK.OUT1 → DigitalOutput_Q1.OUT
ILOCK.OUT2 → DigitalOutput_Q2.OUT
```

Ablauf:

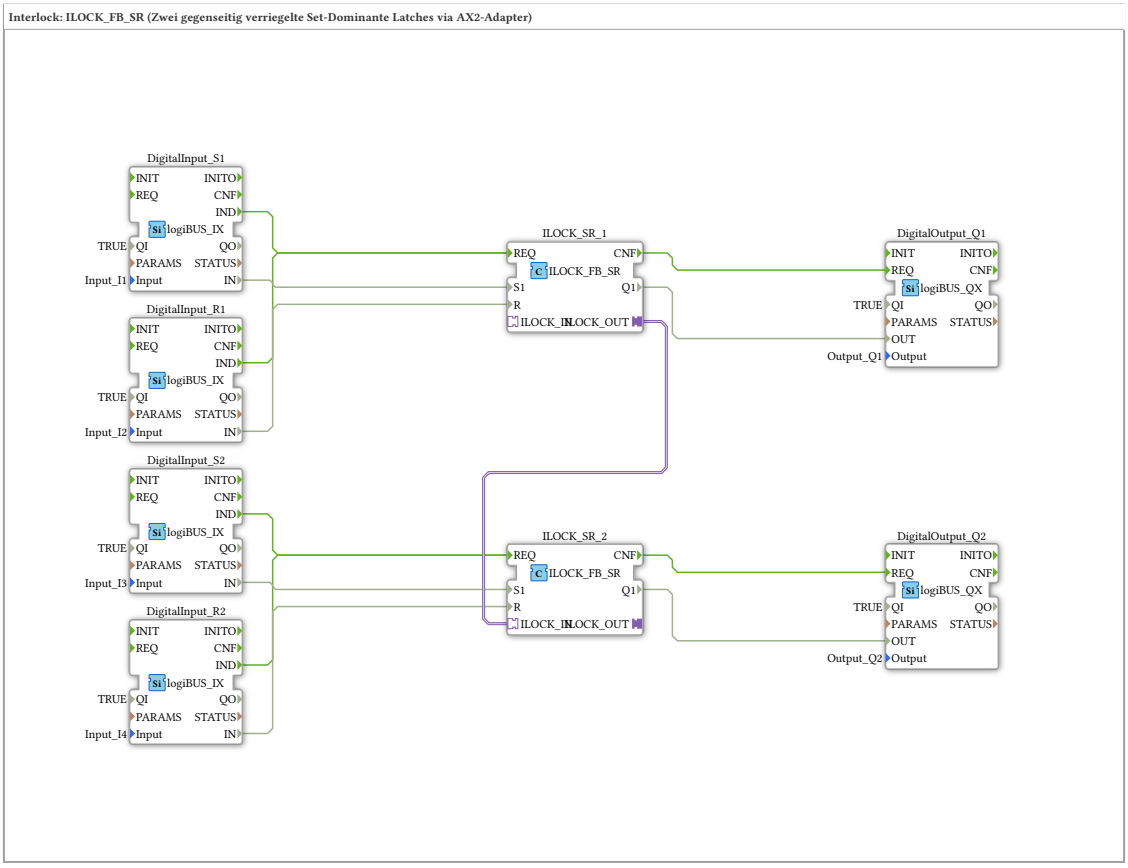
1. **Setzen von Q1**: Ein Tastendruck an Eingang I1 erzeugt ein Ereignis am Ausgang `IND` des Bausteins `DigitalInput_I1`. Dieses Ereignis wird an den Ereigniseingang `CLK1` des Interlock-Bausteins `ILOCK` weitergeleitet. Daraufhin setzt `ILOCK` den Datenausgang `OUT1` auf `TRUE` und `OUT2` auf `FALSE` (gegenseitige Verriegelung). Anschließend wird über den Ereignisausgang `EO` der Ausgangsbaustein `DigitalOutput_Q1` getriggert, der den Wert von `OUT1` an den physikalischen Ausgang Q1 weitergibt. Auch `DigitalOutput_Q2` erhält dasselbe Ereignis und übernimmt den Wert von `OUT2` (der nun `FALSE` ist).
2. **Setzen von Q2**: Analog führt ein Tastendruck an Eingang I2 über `DigitalInput_I2` und den Ereigniseingang `CLK2` zum Setzen von `OUT2` (und Rücksetzen von `OUT1`).
3. **Reset**: Ein Tastendruck an Eingang I3 wird an den Ereigniseingang `R` des Interlock-Bausteins geführt. Dies setzt beide Ausgänge `OUT1` und `OUT2` zurück auf `FALSE`. Über `EO` werden wieder beide Ausgangsbausteine aktualisiert.

Besonderheit: Die Ausgangsbausteine werden bei jedem Ereignis (egal ob Setzen oder Reset) gemeinsam getriggert, sodass beide Ausgänge stets synchron den Zustand des Interlocks abbilden.

Zusammenfassung

Die Übung demonstriert den Einsatz des standardisierten Interlock-Funktionsbausteins `ILOCK_2_E` aus der logiBUS-Bibliothek. Durch die Verwendung von Ereignis-gesteuerten Digital-Eingängen und -Ausgängen wird ein einfaches, aber sicheres Verriegelungssystem aufgebaut. Die gegenseitige Verriegelung stellt sicher, dass nie beide Ausgänge gleichzeitig aktiv werden – eine typische Anforderung in der Automatisierungstechnik.

2.2.1.372 Uebung_208: Interlock: ILOCK_FB_SR (Zwei gegenseitig verriegelte Set-Dominante Latches via AX2-Adapter)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert die Realisierung einer gegenseitigen Verriegelung (Interlock) zwischen zwei Ausgängen mithilfe des Funktionsbausteins `ILOCK_FB_SR`. Jeweils ein Set-dominantes Latch steuert einen Ausgang, wobei über eine Adapterverbindung sichergestellt wird, dass immer nur einer der beiden Ausgänge aktiv sein kann. Die Eingänge (Set und Reset) werden über digitale Eingabebaugruppen (logiBUS_IX) eingelesen, die Ausgänge über digitale Ausgabebaugruppen (logiBUS_QX) ausgegeben. Die Verriegelung verhindert, dass beide Ausgänge gleichzeitig gesetzt werden – selbst wenn beide Set-Signale gleichzeitig anliegen.

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_S1, DigitalInput_R1, DigitalInput_S2, DigitalInput_R2**
  - **Typ**: `logiBUS::io::DI::logiBUS_IX`
  - **Parameter**:
    - `QI` = `TRUE` (Intern aktiviert)
    - `Input` = `Input_I1` (bzw. `I2`, `I3`, `I4`) – Zuordnung zum realen Eingangskanal
  - **Funktion**: Wandelt das binäre Signal des angeschlossenen Sensors in ein digitales Datensignal (`IN`) und erzeugt bei einer Flanke ein Ereignis (`IND`).
- **ILOCK_SR_1, ILOCK_SR_2**
  - **Typ**: `logiBUS::signalprocessing::interlock::ILOCK_FB_SR`
  - **Parameter**: Keine benutzerdefinierten Parameter (werden über Verbindungen konfiguriert).
  - **Funktion**: Set-dominantes Latch mit Verriegelungslogik. Über den Adapter (`ILOCK_IN`/`ILOCK_OUT`) wird die gegenseitige Blockade realisiert. Die Bausteine arbeiten intern wie folgt:
    - `S1` (Set) hat Priorität vor `R` (Reset) – bei aktivem Set wird der Ausgang `Q1` gesetzt, solange der verriegelnde Eingang (`ILOCK_IN`) nicht aktiv ist.

    - R (Reset) setzt Q1 zurück, wenn S1 inaktiv ist.
    - Q1 wird nur dann aktiv, wenn der verriegelnde Partnerausgang (ILOCK_IN von der anderen Instanz) nicht gesetzt ist.
  - **Adapteranschluss**:
    - ILOCK_OUT (von ILOCK_SR_1) ⟶ ILOCK_IN (von ILOCK_SR_2)
    - Dadurch wird verhindert, dass beide Bausteine gleichzeitig Q1 = TRUE setzen.
- **DigitalOutput_Q1, DigitalOutput_Q2**
  - **Typ**: logiBUS::io::DQ::logiBUS_QX
  - **Parameter**:
    - QI = TRUE (Intern aktiviert)
    - Output = Output_Q1 (bzw. Q2) – Zuordnung zum realen Ausgangskanal
  - **Funktion**: Setzt den physikalischen Ausgang auf den über OUT anliegenden Wert, sobald ein Ereignis (REQ) eintrifft.

Programmablauf und Verbindungen

1. **Eingangsverarbeitung** Jeder DigitalInput-FB (logiBUS_IX) wartet auf eine Signaländerung an seinem zugehörigen Hardware-Eingang (Input_I1 ... Input_I4). Bei einer Flanke erzeugt er das Ereignis IND und stellt den aktuellen Zustand am Datenausgang IN bereit.
2. **Verriegelungslogik**
   - Das Ereignis IND des jeweiligen Eingangs wird direkt an den REQ-Eingang des zugehörigen ILOCK_FB_SR weitergeleitet.
   - Gleichzeitig wird der Datenwert IN an den entsprechenden Set- oder Reset-Eingang des ILOCKs angelegt:
     - DigitalInput_S1.IN ⟶ ILOCK_SR_1.S1
     - DigitalInput_R1.IN ⟶ ILOCK_SR_1.R
     - DigitalInput_S2.IN ⟶ ILOCK_SR_2.S1
     - DigitalInput_R2.IN ⟶ ILOCK_SR_2.R
   - Die beiden ILOCK-Bausteine sind über ihre Adapteranschlüsse verriegelt: ILOCK_SR_1.ILOCK_OUT ⟶ ILOCK_SR_2.ILOCK_IN. Dadurch kann ILOCK_SR_2 seinen Ausgang Q1 nur dann auf TRUE setzen, wenn ILOCK_SR_1.Q1 = FALSE ist (und umgekehrt).
3. **Ausgabe**
   - Nach der Verarbeitung erzeugt der ILOCK-Baustein das Ereignis CNF. Dieses triggert den zugehörigen DigitalOutput-FB (logiBUS_QX) über dessen REQ-Eingang.
   - Gleichzeitig wird das Ergebnis Q1 des ILOCKs an den Datenausgang OUT des DigitalOutputs übergeben:
     - ILOCK_SR_1.Q1 ⟶ DigitalOutput_Q1.OUT
     - ILOCK_SR_2.Q1 ⟶ DigitalOutput_Q2.OUT

Lernziele:

- Verständnis der Arbeitsweise eines Set-dominanten Latches mit Interlock-Funktion.
- Umsetzung einer gegenseitigen Verriegelung (z. B. für Schutzfunktionen oder Richtungssteuerungen) mittels Adapterverbindung.
- Einbindung von digitalen Ein-/Ausgabebausteinen in ein logisches Steuerungsnetzwerk.

Schwierigkeitsgrad: Mittel – erfordert Grundkenntnisse in der Signalverarbeitung und der Arbeit mit Funktionsbausteinen in 4diac.

Vorgehen zur Inbetriebnahme:

- Laden Sie die Übung in die 4diac-IDE.
- Stellen Sie sicher, dass die Hardware-Anschlüsse Input_I1...Input_I4 den Tastern/Sensoren für S1, R1, S2, R2 entsprechen und Output_Q1/Q2 die zu steuernden Aktoren ansteuern.
- Starten Sie die Ausführung und testen Sie durch Betätigen der Taster das Setzen und Zurücksetzen der Ausgänge. Dabei sollte nie gleichzeitig Q1 und Q2 TRUE sein.

Zusammenfassung

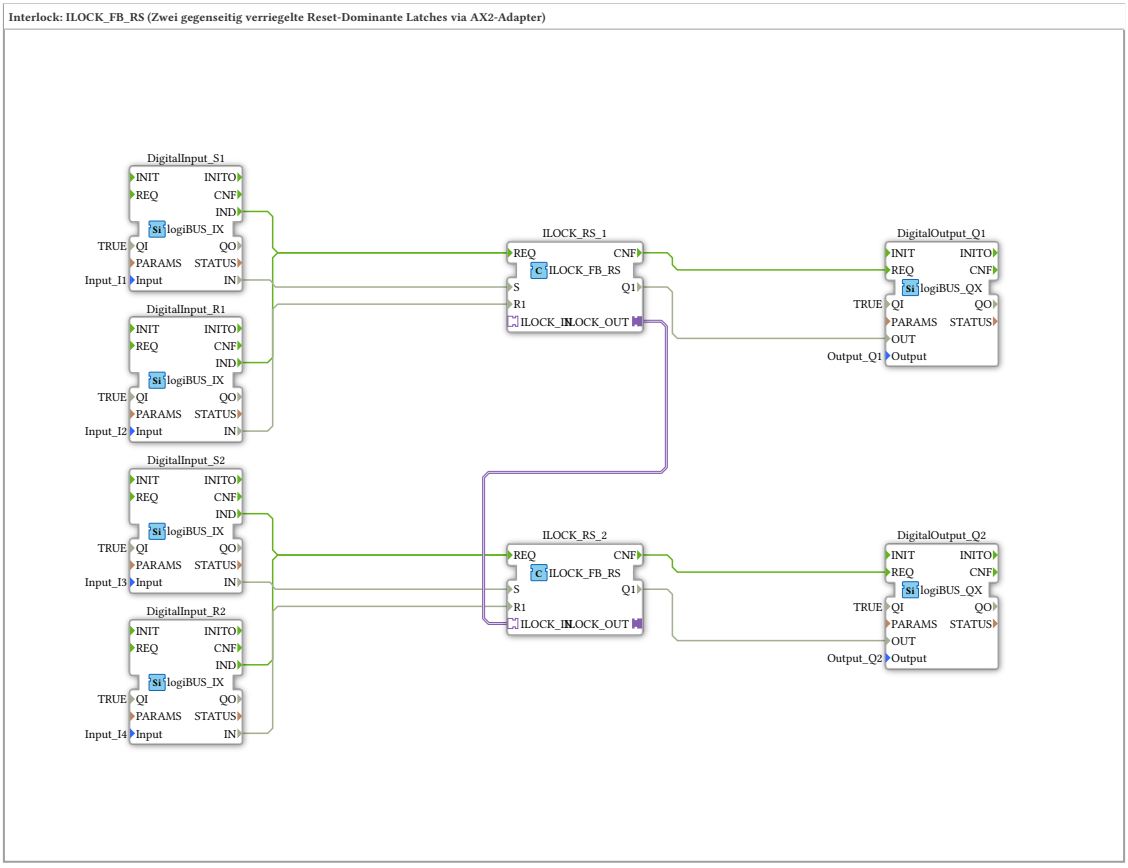
In dieser Übung wurde eine gegenseitige Verriegelung zweier Ausgänge mit dem Funktionsbaustein ILOCK_FB_SR realisiert. Die beiden ILOCK-Bausteine sind über einen Adapter so verbunden, dass nur einer der Ausgänge aktiv sein kann – ein typisches Anwendungsbeispiel für Verriegelungen in der Automati-

sierungstechnik. Die Übung vermittelt den Umgang mit digitalen Ein-/Ausgabemodulen und zeigt, wie sich komplexe Logiken wie Set-Dominanz und Interlocking in einem 4diac-Netzwerk abbilden lassen.

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

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

2.2.1.373 Uebung_209: Interlock: ILOCK_FB_RS (Zwei gegenseitig verriegelte Reset-Dominante Latches via AX2-Adapter)



• – ○ • – ○ • – ○ •

Einleitung

In dieser Übung wird eine **gegenseitige Verriegelung** (Interlock) zwischen zwei reset-dominanten RS-Latches realisiert. Die Funktionsbausteine `ILOCK_FB_RS` sind über einen AX2-Adapter miteinander verbunden, sodass immer nur einer der beiden Ausgänge aktiv sein kann. Sobald ein Latch gesetzt wird, wird der andere zwangsweise zurückgesetzt. Die Ein- und Ausgänge sind an digitale logiBUS-Hardware (Eingänge I1–I4, Ausgänge Q1 und Q2) angebunden.

Die Übung vermittelt den Umgang mit speziellen Interlock-Bausteinen, die in der Steuerungstechnik für wechselseitige Sicherungen (z. B. bei Motoren oder Ventilen) eingesetzt werden.

Verwendete Funktionsbausteine (FBs)

Bausteinname	Typ	Parameter / Anschlüsse
`DigitalInput_S1`	`logiBUS::io::DI::logiBUS_IX`	`QI = TRUE`, `Input = Input_I1` (Einschaltsignal 1)
`DigitalInput_R1`	`logiBUS::io::DI::logiBUS_IX`	`QI = TRUE`, `Input = Input_I2` (Rücksetzsignal 1)

DigitalInput_S2	logiBUS::io::DI::logiBUS_IX	QI = TRUE, Input = Input_I3 (Einschaltsignal 2)
DigitalInput_R2	logiBUS::io::DI::logiBUS_IX	QI = TRUE, Input = Input_I4 (Rücksetzsignal 2)
ILOCK_RS_1	logiBUS::signalprocessing::interlock::ILOCK_FB_RS	–
ILOCK_RS_2	logiBUS::signalprocessing::interlock::ILOCK_FB_RS	–
DigitalOutput_Q1	logiBUS::io::DQ::logiBUS_QX	QI = TRUE, Output = Output_Q1 (Lampe/Signal 1)
DigitalOutput_Q2	logiBUS::io::DQ::logiBUS_QX	QI = TRUE, Output = Output_Q2 (Lampe/Signal 2)

Erläuterung der Interlock-Bausteine ILOCK_FB_RS ist ein reset-dominantes RS-Latch mit einer zusätzlichen Adapterschnittstelle (ILOCK_IN, ILOCK_OUT). Über diese Adapterverbindung können mehrere solche Bausteine gekoppelt werden: Wird ein Latch gesetzt, sendet es ein Signal auf dem ILOCK_OUT-Adapter, das den anderen Baustein über ILOCK_IN in den Reset-Zustand zwingt. Somit ist zu jedem Zeitpunkt maximal einer der beiden Ausgänge Q1 aktiv.

Programmablauf und Verbindungen

Das System ist **ereignisgesteuert** aufgebaut:

1. **Eingangssignal** Ein Signal auf einem Digitaleingang (z. B. Input_I1 für Setzen von Latch 1) erzeugt ein Ereignis IND am entsprechenden DigitalInput-FB.
2. **Ablauf im Latch** Dieses Ereignis wird auf den REQ-Eingang des zugehörigen ILOCK_RS-Bausteins weitergeleitet. Gleichzeitig werden die Datenwerte (S und R1) aus dem Digitaleingang an den Latch übergeben. Der Baustein verarbeitet die Signale (Reset dominant) und gibt bei Abschluss ein CNF-Ereignis aus.
3. **Ausgang** Das CNF-Ereignis aktiviert den DigitalOutput-Baustein, der den aktuellen Zustand des Latches auf den physikalischen Ausgang (z. B. Output_Q1) legt.
4. **Verriegelung** Der Adapterausgang ILOCK_RS_1.ILOCK_OUT ist mit dem Adaptereingang ILOCK_RS_2.ILOCK_IN verbunden. Wenn Latch 1 gesetzt wird, erhält Latch 2 über die Adapterleitung ein aktives Signal, das ihn zurücksetzt (und umgekehrt). Dadurch kann nie gleichzeitig Q1 und Q2 HIGH sein.

Verbindungsübersicht:

Quelle	Ziel	Art
DigitalInput_S1.IND	ILOCK_RS_1.REQ	Ereignis
DigitalInput_R1.IND	ILOCK_RS_1.REQ	Ereignis
DigitalInput_S2.IND	ILOCK_RS_2.REQ	Ereignis
DigitalInput_R2.IND	ILOCK_RS_2.REQ	Ereignis
ILOCK_RS_1.CNF	DigitalOutput_Q1.REQ	Ereignis
ILOCK_RS_2.CNF	DigitalOutput_Q2.REQ	Ereignis
DigitalInput_S1.IN	ILOCK_RS_1.S	Daten
DigitalInput_R1.IN	ILOCK_RS_1.R1	Daten
DigitalInput_S2.IN	ILOCK_RS_2.S	Daten
DigitalInput_R2.IN	ILOCK_RS_2.R1	Daten
ILOCK_RS_1.Q1	DigitalOutput_Q1.OUT	Daten
ILOCK_RS_2.Q1	DigitalOutput_Q2.OUT	Daten

`ILOCK_RS_1.ILOCK_OUT`	`ILOCK_RS_2.ILOCK_IN`	**Adapter** (Interlock)

Zusammenfassung

Die Übung demonstriert eine **wechselseitige Verriegelung** zweier reset-dominanter RS-Latches mithilfe der speziellen `ILOCK_FB_RS`-Bausteine und deren Adapterkopplung. Wichtige Lerninhalte:

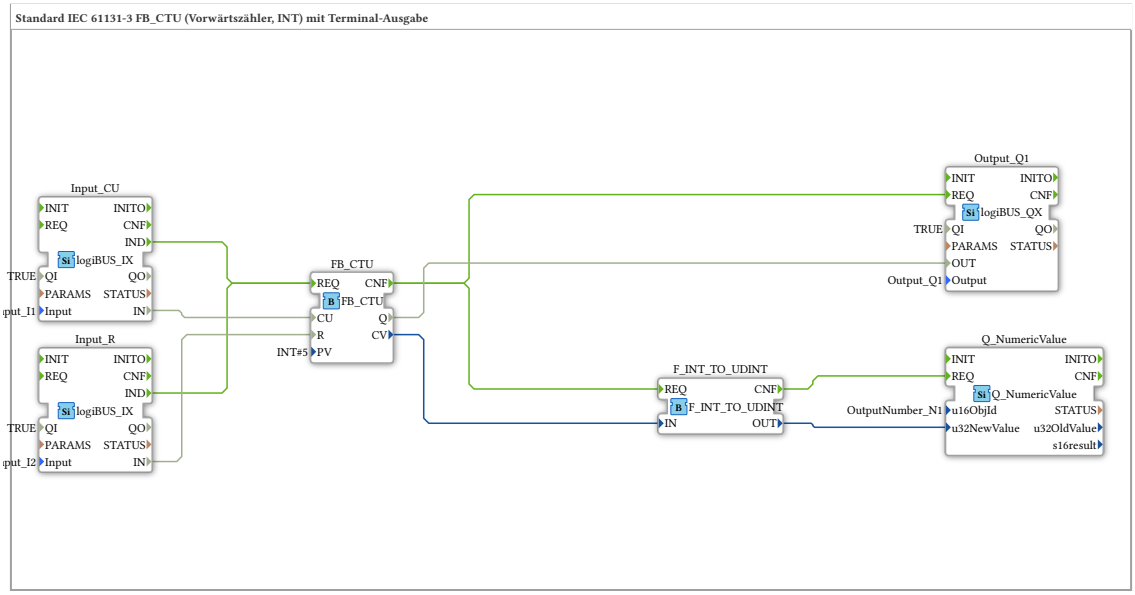
- Verständnis des Interlock-Prinzips (gegenseitige Ausschließlichkeit).
- Anwendung von Adapterverbindungen zur Querkommunikation zwischen Funktionsbausteinen.
- Einbindung digitaler Ein-/Ausgabehardware (logiBUS).
- Ereignisgesteuerter Ablauf in 4diac-FBS.

Nach erfolgreicher Durchführung kann der Anwender solche Verriegelungen in eigenen Steuerungsprojekten einsetzen, z. B. für die Absicherung konkurrierender Aktoren.

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

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

2.2.1.374 Uebung_210: Standard IEC 61131-3 FB_CTU (Vorwärtszähler, INT) mit Terminal-Ausgabe



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung implementiert einen Vorwärtszähler (Count-Up) basierend auf dem Standardbaustein **FB_CTU** nach IEC 61131-3. Der Zähler arbeitet mit einem Datentyp `INT` (16-Bit Ganzzahl) und verfügt über eine Terminal-Ausgabe, die den aktuellen Zählwert numerisch anzeigt. Als Hardware-Anbindung dienen digitale Eingänge und ein digitaler Ausgang des logiBUS-Systems.

Verwendete Funktionsbausteine (FBs)

Bausteinname	Typ	Parameter	Ereigniseingänge/-ausgänge	Dateneingänge/-ausgänge
FB_CTU	`iec61131::counters::FB_CTU`	PV = INT#5	REQ (Eingang), CNF (Ausgang)	CU (Eingang), R (Eingang), Q (Ausgang), CV (Ausgang)

Input_CU	logiBUS::io::DI::logiBUS_DI_X	QI = TRUE, Input = Input_I1	IND (Ausgang)	IN (Ausgang)
Input_R	logiBUS::io::DI::logiBUS_DI_X	QI = TRUE, Input = Input_I2	IND (Ausgang)	IN (Ausgang)
Output_Q1	logiBUS::io::DQ::logiBUS_DQ_X	QO = TRUE, Output = Output_Q1	REQ (Eingang)	OUT (Eingang)
F_INT_TO_UDINT	Type conversion::F_INT_TO_UDINT		REQ (Eingang), CNF (Ausgang)	IN (Eingang), OUT (Ausgang)
Q_NumericValue	Visualisierung::UT::Q::Q_NumericValue	QI = TRUE, OutputNumber_N1	REQ (Eingang)	u32NewValue (Eingang)

Funktionsweise der einzelnen Bausteine

- **FB_CTU**: Der Vorwärtszähler erhöht bei jeder positiven Flanke am Eingang *CU* den internen Zählwert *CV* um 1. Erreicht *CV* den voreingestellten Wert *PV* (hier: 5), wird der Ausgang *Q* gesetzt. Ein Signal am Eingang *R* setzt *CV* zurück auf 0 und *Q* zurück auf FALSE. Der Baustein wird über den Ereigniseingang *REQ* aktiviert.
- **Input_CU** und **Input_R**: Lesen jeweils einen digitalen Hardware-Eingang (logiBUS-Klemme) und geben bei einer Signaländerung ein Ereignis (*IND*) sowie den aktuellen Zustand (*IN*) aus.
- **Output_Q1**: Empfängt ein Ereignis und setzt den angeschlossenen digitalen Ausgang auf den Wert des Dateneingangs *OUT*.
- **F_INT_TO_UDINT**: Wandelt den aktuellen Zählwert *CV* (Datentyp `INT`) in einen vorzeichenlosen 32-Bit Wert (`UDINT`) um, da die nachfolgende Terminal-Ausgabe nur positive Werte verarbeiten kann.
- **Q_NumericValue**: Stellt einen numerischen Wert auf dem Terminal (HMI) dar. Der Wert wird über *u32NewValue* übergeben und das Display über die Objekt-ID *OutputNumber_N1* adressiert.

Programmablauf und Verbindungen

Der Ablauf wird durch Ereignisverbindungen gesteuert:

1. **Zählimpulse**: Tritt eine Änderung am digitalen Eingang *Input_I1* auf, sendet `Input_CU.IND` ein Ereignis an `FB_CTU.REQ`. Gleichzeitig wird der Signalzustand über `Input_CU.IN` → `FB_CTU.CU` weitergeleitet.
2. **Rücksetzen**: Analog löst eine Änderung an *Input_I2* ein Ereignis von `Input_R.IND` aus, das ebenfalls an `FB_CTU.REQ` geht. Der Wert von `Input_R.IN` wird an den Rücksetzeingang `FB_CTU.R` geführt.
3. **Ausgang setzen**: Nach jedem Bearbeitungsschritt des Zählers (Ereignisausgang `FB_CTU.CNF`) werden zwei Aktionen parallel ausgelöst:
   - Der Ausgangswert *Q* wird über `FB_CTU.Q` → `Output_Q1.OUT` an den digitalen Ausgang *Output_Q1* übergeben und durch `Output_Q1.REQ` ausgegeben.
   - Der aktuelle Zählwert *CV* wird über `FB_CTU.CV` → `F_INT_TO_UDINT.IN` gewandelt. Der gewandelte `UDINT`-Wert (`F_INT_TO_UDINT.OUT`) wird an `Q_NumericValue.u32NewValue` übergeben. Ein weiteres Ereignis (`F_INT_TO_UDINT.CNF`) aktiviert `Q_NumericValue.REQ` zur Aktualisierung der Terminalanzeige.

Hinweise aus dem Entwurf:

- Die Umwandlung von `INT` nach `UDINT` ist nicht optimal, da negative Zählwerte nicht abgebildet werden können. Eine Alternative könnte die Verwendung eines vorzeichenbehafteten Datentyps oder eines anderen Ausgabebausteins sein.
- Da beide Eingänge (`Input_CU` und `Input_R`) dasselbe Ereignis an `FB_CTU.REQ` senden, kann es zu einer hohen Ereignisfrequenz kommen. In der Praxis sollte hier ein **E_D_FF** (Ereignis-D-Flipflop) oder ein ähnlicher Entprellmechanismus zwischengeschaltet werden, um unnötige Verarbeitungen zu vermeiden.

Zusammenfassung

Diese Übung vermittelt den praktischen Umgang mit dem IEC-61131-3 Zählerbaustein **FB_CTU** in der 4diac-IDE. Lernziele sind:

- Erstellung eines einfachen Vorwärtszählers mit einstellbarem Schwellwert.
- Anbindung digitaler Ein- und Ausgänge über den logiBUS.

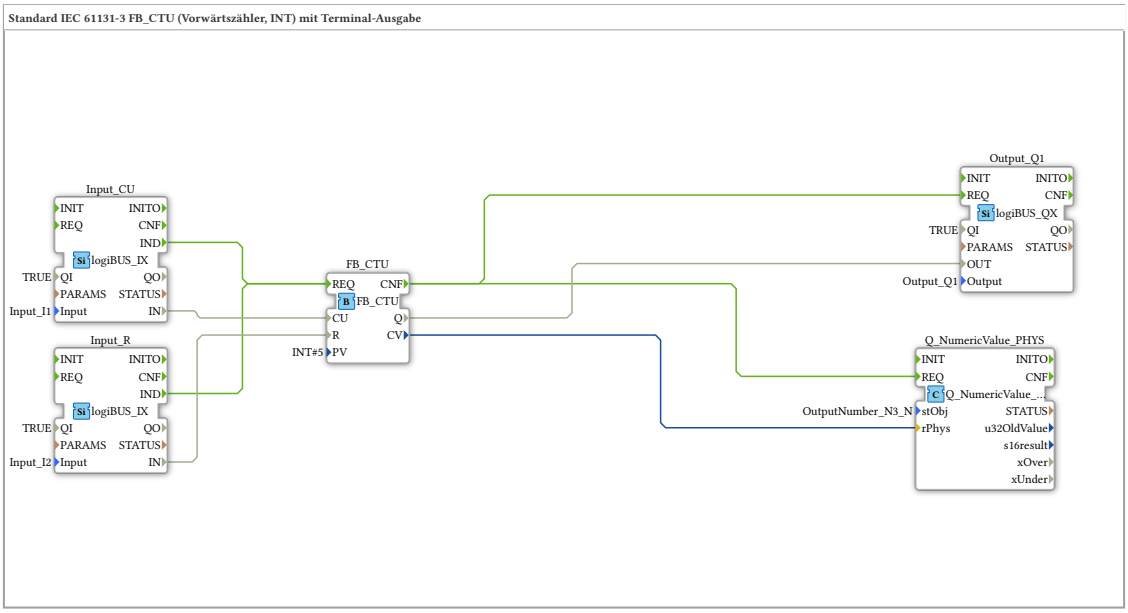
- Datenkonvertierung und Darstellung numerischer Werte auf einem Terminal.
- Verständnis der ereignisgesteuerten Ausführung und der Notwendigkeit von Ereignisreduzierung (z. B. durch Flipflops).

Schwierigkeitsgrad: Mittel **Vorkenntnisse**: Grundlegende Bedienung der 4diac-IDE, Verständnis von Ereignissen und Datenverbindungen, Basiswissen über IEC-61131-3 Funktionsbausteine. **Start der Übung**: Importieren Sie die Datei `Uebung_210.fbt` (oder das entsprechende 4diac-Projekt) und weisen Sie die logiBUS-Hardware-Kanäle entsprechend den Pool-Bezeichnungen (`Input_I1`, `Input_I2`, `Output_Q1`, `OutputNumber_N1`) zu.

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

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

2.2.1.375 Uebung_210b: Standard IEC 61131-3 FB_CTU (Vorwärtszähler, INT) mit Terminal-Ausgabe



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung implementiert einen **Vorwärtszähler (Counter Up)** nach IEC 61131-3 (FB_CTU) mit dem Datentyp `INT`. Der Zähler wird über zwei digitale Eingänge gesteuert: ein Zählimpuls-Eingang (`CU`) und ein Rücksetz-Eingang (`R`). Der aktuelle Zählerstand wird sowohl auf einen digitalen Ausgang (Grenzwert erreicht) als auch über ein Terminal-Objekt zur numerischen Anzeige ausgegeben.

Die Übung demonstriert die grundlegende Funktionsweise eines industriellen Zählers, die Einbindung von Hardware-Ein-/Ausgängen (logiBUS) sowie die Ausgabe von Werten auf eine numerische Anzeige (Terminal).

Verwendete Funktionsbausteine (FBs)

In dieser Übung werden folgende Funktionsbausteine eingesetzt:

FB-Name	Typ	Parameter	Kurzbeschreibung
`FB_CTU`	`iec61131::counters::FB_CTU`	`PV = INT#5`	IEC 61131-3 Vorwärtszähler, Zählbereich INT, Preset-Wert 5.

`Input_CU`	`logiBUS::io::DI::logiBUS_IO_DI`	`stObj := TRUE, Input := Input_I1`	Digitaler Eingang, liefert den Zählimpuls (`CU`).
`Input_R`	`logiBUS::io::DI::logiBUS_IO_DI`	`stObj := TRUE, Input := Input_I2`	Digitaler Eingang, liefert das Rücksetzsignal (`R`).
`Output_Q1`	`logiBUS::io::DQ::logiBUS_IO_DQ`	`stObj := TRUE, Output := Output_Q1`	Digitaler Ausgang, wird aktiviert wenn der Zähler seinen Endwert erreicht hat (`Q`).
`Q_NumericValue_PHYS`	`PHYBUS::UT::Q::Q_NumericValue_PHYS`	`stObj := OutputNumber_N3`	Terminal-Ausgabe: Zeigt den aktuellen Zählerstand (CV) numerisch an.

Hinweise zu den Hardware-FBs: Die Eingänge `Input_I1` und `Input_I2` sowie der Ausgang `Output_Q1` sind physische logiBUS-Kanäle. Das Terminal-Objekt `OutputNumber_N3` ist ein vordefiniertes numerisches Anzeigeelement, das den Zählerwert darstellt.

Programmablauf und Verbindungen

Ereignis- und Datenverbindungen

Die nachfolgende Grafik zeigt die logischen Verbindungen zwischen den Bausteinen (basierend auf dem XML-Netzwerk):

```
[Input_Cu]  ── IND ─→ REQ [FB_CTU]
[Input_R ]  ── IND ─→ REQ [FB_CTU]
[FB_CTU ]  ── CNF ─→ REQ [Output_Q1]
[FB_CTU ]  ── CNF ─→ REQ [Q_NumericValue_PHYS]

Daten:
[Input_Cu.IN]  ──→ FB_CTU.CU
[Input_R.IN]   ──→ FB_CTU.R
[FB_CTU.Q]     ──→ Output_Q1.OUT
[FB_CTU.CV]    ──→ Q_NumericValue_PHYS.rPhys
```

Erklärung:

- **Zählereingänge**: Die beiden digitalen Eingänge `Input_CU` und `Input_R` werden über ihre `IND`-Ereignisse mit dem `REQ`-Eingang des Zählers `FB_CTU` verbunden. Dadurch wird der Zähler bei jeder positiven Flanke der Eingänge bearbeitet. Der Datenwert des jeweiligen Eingangs (`IN`) wird auf den entsprechenden Zähleingang (`CU` bzw. `R`) gelegt.
- **Zählerverhalten**: Der `FB_CTU` zählt bei jeder steigenden Flanke an `CU` hoch. Der aktuelle Zählerstand ist auf dem Datenausgang `CV` verfügbar. Ist der Zählerstand größer oder gleich dem Preset-Wert `PV` (hier `INT#5`), wird der Ausgang `Q` auf `TRUE` gesetzt. Ein `TRUE`-Signal an `R` setzt den Zähler zurück (CV = 0, Q = FALSE).
- **Ausgabe**: Nach jedem Zählvorgang wird das `CNF`-Ereignis des Zählers an den digitalen Ausgang `Output_Q1` und an die Terminal-Ausgabe `Q_NumericValue_PHYS` weitergeleitet.
  - Der `Q`-Wert wird auf den Ausgang `Output_Q1` geschrieben.
  - Der `CV`-Wert wird als physikalische Größe (`rPhys`) an das Terminal übergeben und dort numerisch dargestellt.

Anmerkungen aus dem Quellcode

- Der Kommentar „**INT kann ohne Konvertierung auf REAL geschlossen werden**“ bezieht sich darauf, dass der Zählerwert vom Typ `INT` direkt an den `rPhys`-Eingang (Typ `REAL`) angeschlossen werden kann – eine automatische Typkonvertierung findet statt.
- Der Hinweis „**hier gegebenenfalls einen E_D_FF einbauen, damit die Events reduziert werden**“ empfiehlt, bei schnellen Impulsen einen Flanken-Detektor vorzuschalten, um Fehlauslösungen zu vermeiden.

- Der Kommentar „**F_INT_TO_REAL kann man weglassen**“ bestätigt die direkte Konvertierung ohne expliziten Baustein.

Lernziele und Vorkenntnisse

- **Lernziele**:
  - Einbindung eines IEC-61131-3-Zählers in eine 4diac-Applikation.
  - Verknüpfung digitaler Ein- und Ausgänge mit logiBUS-Hardware.
  - Ausgabe von numerischen Werten auf einem Terminal.
- **Schwierigkeitsgrad**: Einfach
- **Vorkenntnisse**: Grundlegende Kenntnisse der 4diac-IDE, Verständnis von Ereignis- und Datenverbindungen.
- **Start der Übung**: Die Übung kann direkt in einer laufenden 4diac-Umgebung mit angeschlossener logiBUS-Hardware ausgeführt werden. Die Eingänge `Input_I1` (Taster) und `Input_I2` (Taster) steuern den Zähler; `Output_Q1` kann z.B. eine Lampe ansteuern.

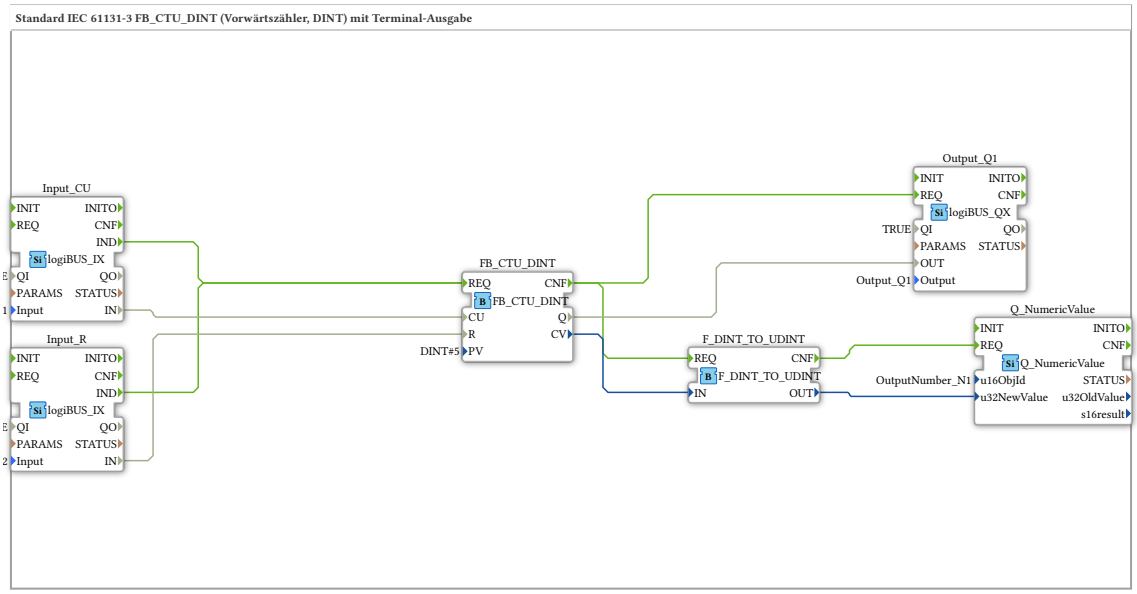
Zusammenfassung

Die Übung **Uebung_210b** realisiert einen vollständigen IEC-61131-3 Vorwärtszähler (`FB_CTU`) mit zwei digitalen Steuereingängen und einer Ausgabe des Zählerstands auf ein Terminal. Der Preset-Wert ist auf 5 gesetzt. Die Applikation verdeutlicht die Verbindung von Hardware-E/A mit einem Standardfunktionsbaustein und die unkomplizierte Datenausgabe mittels Terminal-Objekt. Sie eignet sich als Einstiegsübung in die industrielle Zählerprogrammierung mit 4diac.

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

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

2.2.1.376 Uebung_211: Standard IEC 61131-3 FB_CTU_DINT (Vorwärtszähler, DINT) mit Terminal-Ausgabe



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung implementiert einen Vorwärtszähler (Counter Up) nach IEC 61131-3 mit dem Funktionsbaustein `FB_CTU_DINT`. Der Zähler verwendet den Datentyp `DINT` (doppelt genaue Ganzzahl). Die Zählimpulse werden über einen digitalen Eingang (I1) bereitgestellt, ein zweiter digitaler Eingang (I2) dient zum Rücksetzen des Zählers. Der Ausgang des Zählers (Q) steuert einen digitalen Ausgang (Q1), gleichzeitig wird der aktuelle Zählerwert (CV) über eine Terminal-Ausgabe auf einem Display sichtbar gemacht.

Die Übung zeigt die grundlegende Verschaltung von industriellen Ein-/Ausgangsbausteinen (logiBUS) mit einem Zählbaustein und einer numerischen Anzeige. Ein Kommentar in der Schaltung weist darauf

hin, dass die verwendete Datentypkonvertierung `DINT_TO_UDINT` problematisch ist, da negative Zählerstände nicht korrekt dargestellt werden können.

Verwendete Funktionsbausteine (FBs)

- **FB_CTU_DINT**
  Typ: `iec61131::counters::FB_CTU_DINT`
  Parameter: `PV` = `DINT#5` (Voreinstellungswert – der Zähler gibt Q aus, wenn CV >= PV)
  Ereignisseingang: `REQ` – aktiviert die Zählerlogik
  Datenverbindungen:
  - `CU` (Count Up) vom Baustein `Input_CU.IN`
  - `R` (Reset) vom Baustein `Input_R.IN`
  - `Q` (Ausgang) geht an `Output_Q1.OUT`
  - `CV` (aktueller Zählerwert) geht an `F_DINT_TO_UDINT.IN`
- **Input_CU**
  Typ: `logiBUS::io::DI::logiBUS_IX`
  Parameter: `QI` = `TRUE`, `Input` = `Input_I1` (erster digitaler Eingang)
  Ereignisausgang: `IND` – löst bei Flankenänderung aus
  Datenausgang: `IN` – liefert den aktuellen Eingangszustand
- **Input_R**
  Typ: `logiBUS::io::DI::logiBUS_IX`
  Parameter: `QI` = `TRUE`, `Input` = `Input_I2` (zweiter digitaler Eingang)
  Ereignisausgang: `IND` – löst bei Flankenänderung aus
  Datenausgang: `IN` – liefert den aktuellen Eingangszustand
- **Output_Q1**
  Typ: `logiBUS::io::DQ::logiBUS_QX`
  Parameter: `QI` = `TRUE`, `Output` = `Output_Q1`
  Ereigniseingang: `REQ` – aktiviert die Ausgabe
  Dateneingang: `OUT` – erhält den Zustand vom Zählerausgang `Q`
- **Q_NumericValue**
  Typ: `isobus::UT::Q::Q_NumericValue`
  Parameter: `u16ObjId` = `OutputNumber_N1` (Kennung des Terminal-Ausgabefelds)
  Ereigniseingang: `REQ` – aktualisiert die Anzeige
  Dateneingang: `u32NewValue` – neuer Wert zur Anzeige (erwartet `UDINT`)
- **F_DINT_TO_UDINT**
  Typ: `iec61131::conversion::F_DINT_TO_UDINT`
  Ereigniseingang: `REQ` – führt die Konvertierung aus
  Dateneingang: `IN` (`DINT`)
  Datenausgang: `OUT` (`UDINT`)
  Hinweis: Die Konvertierung ist nicht geeignet für negative `DINT`-Werte, da `UDINT` nur positive Zahlen darstellen kann. Ein Kommentar im Netzwerk bezeichnet dies als „großen Quatsch“.

Programmablauf und Verbindungen

Das Zusammenspiel der Bausteine erfolgt wie folgt:

1. **Eingangserfassung:**
   Die beiden digitalen Eingänge `Input_CU` und `Input_R` detektieren Flanken an den physikalischen Anschlüssen I1 und I2. Bei jeder Zustandsänderung senden sie ein Ereignis (`IND`) aus.
2. **Zählerlogik:**
   Die Ereignisse von `Input_CU` und `Input_R` werden auf den **gleichen** Ereigniseingang `FB_CTU_DINT.REQ` geführt. Der Zähler unterscheidet intern anhand der Datenleitungen, ob ein Zählimpuls (`CU`) oder ein Rücksetzen (`R`) angefordert wird.
   Wichtig: Da beide Ereignisquellen direkt auf `REQ` verdrahtet sind, kann es zu Konflikten kommen, wenn beide Eingänge gleichzeitig schalten. Ein Kommentar schlägt vor, einen „E_D_FF“ (Ereignis-D-Flipflop) einzubauen, um die Ereignisse zu reduzieren oder zu priorisieren.

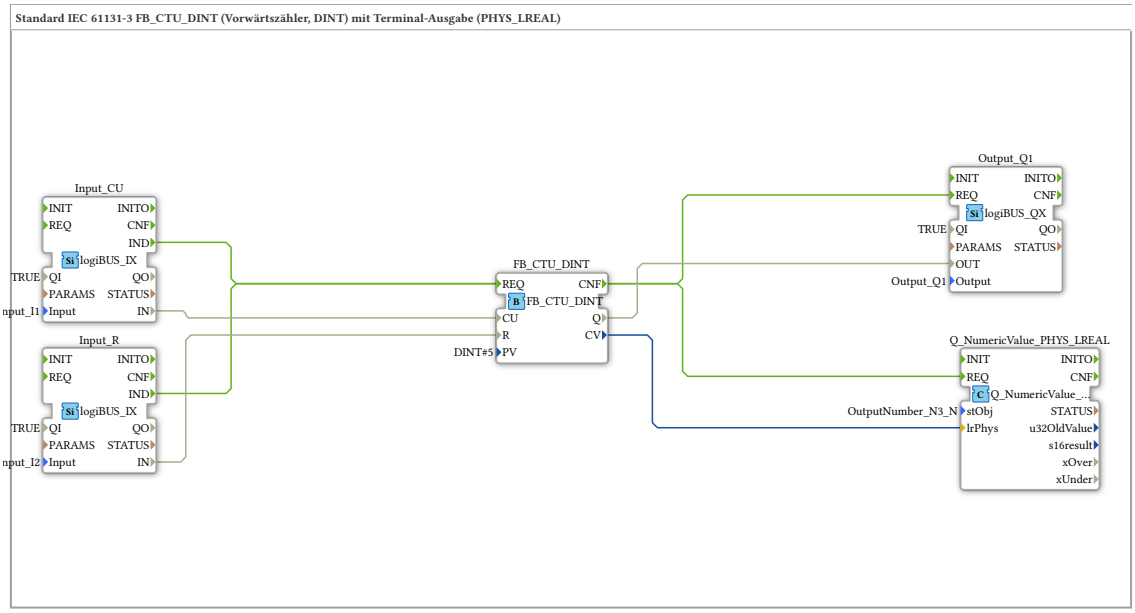
3. **Ausgangssteuerung:**
   Nach der Bearbeitung des Zählers wird das Ereignis `CNF` ausgelöst. Dieses triggert zwei Aktionen parallel:
   - `Output_Q1.REQ` – der digitale Ausgang Q1 wird durch den Wert von `FB_CTU_DINT.Q` gesetzt.
   - `F_DINT_TO_UDINT.REQ` – der aktuelle Zählerstand (CV) wird von `DINT` nach `UDINT` konvertiert.
4. **Terminalausgabe:**
   Die konvertierte Zahl (`UDINT`) wird dem Baustein `Q_NumericValue` übergeben, der nach dem Ereignis `CNF` der Konvertierung getriggert wird. Dadurch wird der Wert auf dem Terminal (z. B. Bedienpanel) mit der Objektkennung `OutputNumber_N1` aktualisiert.

Datentyp-Problematik:
Der Zähler arbeitet mit `DINT` (vorzeichenbehaftet). Bei Überlauf oder Rücksetzen kann der aktuelle Wert negativ werden. Die Konvertierung `DINT_TO_UDINT` interpretiert negative Bitmuster jedoch als sehr große positive Zahlen (z. B. −1 wird zu 4294967295). Dies führt zu einer sinnlosen Anzeige. In der Praxis sollte entweder ein anderer Datentyp verwendet oder eine Begrenzung eingebaut werden.

Zusammenfassung

In dieser Übung wird ein IEC-61131-3 Vorwärtszähler mit zwei digitalen Eingängen (Zählimpuls und Reset) und einem digitalen Ausgang aufgebaut. Der aktuelle Zählerstand wird zusätzlich auf einem Terminal ausgegeben. Der Schaltplan demonstriert die Verwendung von logiBUS-Ein-/Ausgangsbausteinen, einen Konvertierungsbaustein und eine numerische Anzeigekomponente. Gleichzeitig werden typische Fallstricke bei der Datentypkonvertierung aufgezeigt – ein wichtiger Aspekt für die industrielle Steuerungstechnik. Die Übung eignet sich für Einsteiger, die erste Erfahrungen mit Zählfunktionen und der Verschaltung von Ereignis- und Datenverbindungen sammeln möchten.

2.2.1.377 Uebung_211b: Standard IEC 61131-3 FB_CTU_DINT (Vorwärtszähler, DINT) mit Terminal-Ausgabe (PHYS_LREAL)



• – ○ • – ○ • – ○ •

Einleitung

In dieser Übung wird der IEC 61131-3 Standard-Funktionsbaustein **FB_CTU_DINT** (Vorwärtszähler mit `DINT`-Datentyp) verwendet. Der Zähler wird durch einen Taster an einem Digitaleingang erhöht und über einen zweiten Taster zurückgesetzt. Der aktuelle Zählerstand wird zusätzlich über einen Terminal-Ausgabebaustein als physikalischer `LREAL`-Wert ausgegeben.

Ziel ist es, das Zusammenspiel zwischen einem Zähler, digitalen Ein-/Ausgängen und einer numerischen Terminalausgabe in der 4diac-IDE zu verstehen.

Verwendete Funktionsbausteine (FBs)

Im SubApp-Netzwerk werden fünf Instanzen von vordefinierten Bausteintypen verwendet. Es sind keine weiteren Sub-Bausteine definiert.

Name	Typ	Beschreibung
`Input_CU`	`logiBUS_IX`	Digitaler Eingang (logiBUS), der den Zählimpuls (CU) bereitstellt.
`Input_R`	`logiBUS_IX`	Digitaler Eingang, der das Rücksetzsignal (R) bereitstellt.
`FB_CTU_DINT`	`FB_CTU_DINT`	Vorwärtszähler (Counter Up) mit `DINT`-Datentyp.
`Output_Q1`	`logiBUS_QX`	Digitaler Ausgang (logiBUS), der den Zählerausgang Q anzeigt.
`Q_NumericValue_PHYS_LREAL`	`Q_NumericValue_PHYS_LREAL`	Terminal-Ausgabebaustein zur Darstellung eines physikalischen `LREAL`-Wertes.

Detaillierte Parameter und Verbindungen

- **Input_CU**
  - Parameter: `QI` = `TRUE`, `Input` = `Input_I1`
  - Ereignisausgang: `IND` (Signalisiert eine steigende Flanke am Eingang)
  - Datenausgang: `IN` (logischer Zustand des Eingangs)
- **Input_R**
  - Parameter: `QI` = `TRUE`, `Input` = `Input_I2`
  - Ereignisausgang: `IND`
  - Datenausgang: `IN`
- **FB_CTU_DINT**
  - Parameter: `PV` = `DINT#5` (Preset Value = 5)
  - Ereigniseingang: `REQ` (auslösendes Ereignis für Zählvorgang)
  - Dateneingänge: `CU` (Count Up), `R` (Reset)
  - Ereignisausgang: `CNF` (Bestätigung nach Ausführung)
  - Datenausgänge: `Q` (Ausgangssignal – wird TRUE wenn CV ≥ PV), `CV` (aktueller Zählerwert, Typ `DINT`)
- **Output_Q1**
  - Parameter: `QI` = `TRUE`, `Output` = `Output_Q1`
  - Ereigniseingang: `REQ`
  - Dateneingang: `OUT` (Wert, der auf den physikalischen Ausgang geschrieben wird)
- **Q_NumericValue_PHYS_LREAL**
  - Parameter: `stObj` = `OutputNumber_N3` (Referenz auf das Terminal-Ausgabeobjekt)
  - Ereigniseingang: `REQ`
  - Dateneingang: `lrPhys` (physikalischer `LREAL`-Wert zur Anzeige)

Programmablauf und Verbindungen

Ereignisverbindungen

1. **Eingangssignale**
   - Wenn der Taster an `Input_I1` betätigt wird (steigende Flanke), sendet `Input_CU.IND` ein Ereignis an `FB_CTU_DINT.REQ`.
   - Wenn der Taster an `Input_I2` betätigt wird, sendet `Input_R.IND` ebenfalls ein Ereignis an denselben `REQ`-Eingang des Zählers.

   Hinweis: Beide Ereignisse werden auf den gleichen `REQ`-Eingang geführt, daher muss der Baustein intern unterscheiden, welcher Eingang (CU oder R) aktiv ist.

2. **Zählerausführung** Nachdem der Zähler das Ereignis verarbeitet hat (Ausführung der Funktion), sendet er über `CNF` zwei gleichzeitige Ereignisse:
   - an `Output_Q1.REQ` (Aktualisierung des digitalen Ausgangs)
   - an `Q_NumericValue_PHYS_LREAL.REQ` (Aktualisierung der Terminal-Anzeige)

Datenverbindungen

- `Input_CU.IN` ⟶ `FB_CTU_DINT.CU` – Logischer Zustand des Tasters I1 als Zählimpuls.
- `Input_R.IN` ⟶ `FB_CTU_DINT.R` – Logischer Zustand des Tasters I2 als Rücksetzsignal.
- `FB_CTU_DINT.Q` ⟶ `Output_Q1.OUT` – Gibt den Status des Zählausgangs (TRUE, wenn Zählerstand ≥ PV) an den digitalen Ausgang Q1 weiter.
- `FB_CTU_DINT.CV` ⟶ `Q_NumericValue_PHYS_LREAL.lrPhys` – Der aktuelle Zählerstand (`DINT`) wird in einen physikalischen `LREAL`-Wert umgewandelt und auf dem Terminal angezeigt.

Ablauf im Detail

- **Initialzustand:** Zählerstand = 0, Ausgang Q = FALSE, Terminal zeigt 0.0.
- **Zählvorgang:** Jede positive Flanke an `Input_I1` (bei steigender Flanke) erhöht den Zählerstand um 1.
- **Rücksetzen:** Eine positive Flanke an `Input_I2` setzt den Zählerstand auf 0 zurück.
- **Ausgang:** Sobald der Zählerstand den Wert 5 (PV) erreicht oder überschreitet, wird `Q` = TRUE und der digitale Ausgang Q1 schaltet ein.
- **Terminal:** Nach jeder Änderung des Zählerstands wird der neue Wert als `LREAL` auf dem konfigurierten Terminal-Objekt `OutputNumber_N3` angezeigt.

Lernziele

- Verwendung eines IEC 61131-3 Standard-Zählers (FB_CTU_DINT).
- Umgang mit digitalen Ein- und Ausgängen (logiBUS).
- Ausgabe numerischer Werte über einen Terminal-Baustein.
- Verständnis der Ereignis- und Datenflussmodellierung in 4diac.

Benötigte Vorkenntnisse

- Grundlagen der 4diac-IDE (Erstellung von SubApp-Typen, Verbindung von Bausteinen).
- Einfaches Verständnis von Zählern und Binärsignalen.

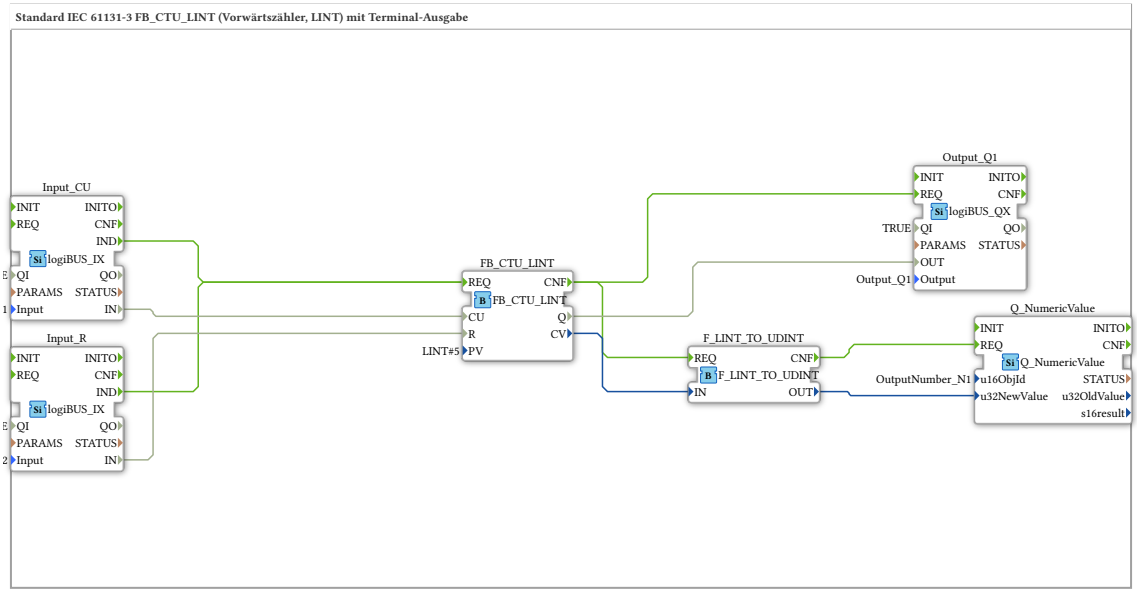
Zusammenfassung

Die Übung 211b demonstriert den praktischen Einsatz eines Vorwärtszählers (CTU) nach IEC 61131-3 in der 4diac-Umgebung. Zwei Taster steuern den Zähler (Zählen und Rücksetzen), der sowohl einen digitalen Ausgang als auch eine numerische Terminalanzeige ansteuert. Der Aufbau zeigt typische Verbindungsmuster für ereignisgesteuerte Automatisierungssoftware und erlaubt ein vertieftes Verständnis der Bausteinparameter, Ereignisverkettungen und Datenkonvertierungen.

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

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

2.2.1.378 Uebung_212: Standard IEC 61131-3 FB_CTU_LINT (Vorwärtszähler, LINT) mit Terminal-Ausgabe



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert die Verwendung des IEC-61131-3-Funktionsbausteins **FB_CTU_LINT** (Vorwärtszähler für große Integer-Werte) in einer 4diac-IDE Subapplikation. Der Zählerstand wird über eine Konvertierung auf einem Terminal ausgegeben. Zusätzlich wird ein digitaler Ausgang gesetzt, sobald der Zähler den vorgegebenen Maximalwert erreicht hat.

Verwendete Funktionsbausteine

Es werden ausschließlich vordefinierte Bibliotheksbausteine eingesetzt; weitere Sub-Bausteine sind nicht enthalten.

- **FB_CTU_LINT**
  - **Typ**: `iec61131::counters::FB_CTU_LINT`
  - **Parameter**: `PV = LINT#5` (Preset-Wert = 5)
  - **Aufgabe**: Vorwärtszähler mit LINT-Datentyp; zählt bei jeder positiven Flanke am CU-Eingang und setzt den Ausgang Q, wenn der Zählerstand den Presetwert erreicht oder überschreitet.
- **Input_CU**
  - **Typ**: `logiBUS::io::DI::logiBUS_IX`
  - **Parameter**: `QI = TRUE`, `Input = Input_I1`
  - **Aufgabe**: Digitaler Eingang, der den Zählimpuls (Clock Up) für den Zähler bereitstellt.
- **Input_R**
  - **Typ**: `logiBUS::io::DI::logiBUS_IX`
  - **Parameter**: `QI = TRUE`, `Input = Input_I2`
  - **Aufgabe**: Digitaler Eingang zum Rücksetzen (Reset) des Zählers.
- **Output_Q1**
  - **Typ**: `logiBUS::io::DQ::logiBUS_QX`
  - **Parameter**: `QI = TRUE`, `Output = Output_Q1`
  - **Aufgabe**: Digitaler Ausgang, der aktiv wird, sobald der Zähler den Presetwert erreicht hat (Q-Signal des Zählers).
- **F_LINT_TO_UDINT**
  - **Typ**: `iec61131::conversion::F_LINT_TO_UDINT`
  - **Parameter**: keiner (Standardkonvertierung)

  - **Aufgabe**: Wandelt den aktuellen Zählerstand (LINT) in einen vorzeichenlosen Doppel-Integer (UDINT) um, da der Terminalbaustein nur positive Werte darstellen kann.
  - **Hinweis**: Ein Kommentar im Netzwerk weist darauf hin, dass diese Konvertierung problematisch ist, weil negative Zahlen nicht dargestellt werden können. Hier könnte man stattdessen einen E_D_FF einbauen, um unerwünschte Ereignisse zu reduzieren.

- **Q_NumericValue**
  - **Typ**: `isobus::UT::Q::Q_NumericValue`
  - **Parameter**: `u16ObjId = OutputNumber_N1`
  - **Aufgabe**: Gibt den übergebenen Zahlenwert auf einem angeschlossenen Terminal (z. B. Isobus-Display) aus. Der Wert wird über den Dateneingang `u32NewValue` aktualisiert.

Programmablauf und Verbindungen

1. **Ereignissteuerung**
   - Jede steigende Flanke an `Input_I1` (verbunden mit `Input_CU`) erzeugt ein Ereignis `IND`, das den Zählerbaustein über den Ereigniseingang `REQ` aktiviert.
   - Jede steigende Flanke an `Input_I2` (verbunden mit `Input_R`) erzeugt ebenfalls ein Ereignis `IND` und aktiviert den Zählerbaustein über denselben `REQ`-Eingang.
   - Nach Abschluss der Zähleroperation signalisiert der Zähler dies über `CNF`. Dieses Ereignis wird an drei Bausteine weitergeleitet:
     - **Output_Q1.REQ** (setzt den digitalen Ausgang entsprechend dem Zähler-Q-Signal)
     - **F_LINT_TO_UDINT.REQ** (startet die Konvertierung des Zählerstands)
     - **Q_NumericValue.REQ** (wird nach erfolgreicher Konvertierung über die Kette `F_LINT_TO_UDINT.CNF → Q_NumericValue.REQ` ausgelöst).
2. **Datenverbindungen**
   - `Input_CU.IN` ⟶ `FB_CTU_LINT.CU` (Zählimpuls)
   - `Input_R.IN` ⟶ `FB_CTU_LINT.R` (Rücksetzen)
   - `FB_CTU_LINT.Q` ⟶ `Output_Q1.OUT` (Ausgangssignal bei Erreichen des Presetwerts)
   - `FB_CTU_LINT.CV` (aktueller Zählerstand) ⟶ `F_LINT_TO_UDINT.IN`
   - `F_LINT_TO_UDINT.OUT` (konvertierter Wert) ⟶ `Q_NumericValue.u32NewValue` (Anzeige auf dem Terminal)
3. **Funktionsweise des Zählers**
   - Der Zähler inkrementiert bei jeder ansteigenden Flanke am CU-Eingang, solange kein Reset erfolgt.
   - Wird der Reset-Eingang auf TRUE gesetzt, wird der Zählerstand auf Null zurückgesetzt.
   - Sobald der Zählerstand den Wert **PV = 5** erreicht, wird der Ausgang `Q` auf TRUE gesetzt (aktives High).
   - Der aktuelle Zählerstand wird kontinuierlich an das Terminal gesendet, sobald sich der Wert ändert.
4. **Lernziele / Schwierigkeit**
   - Einführung in die IEC-61131-3-Zählerfamilie
   - Kombination von digitalen Ein-/Ausgängen mit einem Zähler
   - Datenkonvertierung und Terminalausgabe
   - Schwierigkeitsgrad: **Mittel**
   - Vorkenntnisse: Grundlegendes Verständnis von Ereignis- und Datenflüssen in 4diac sowie einfache IEC-61131-3-Datentypen.

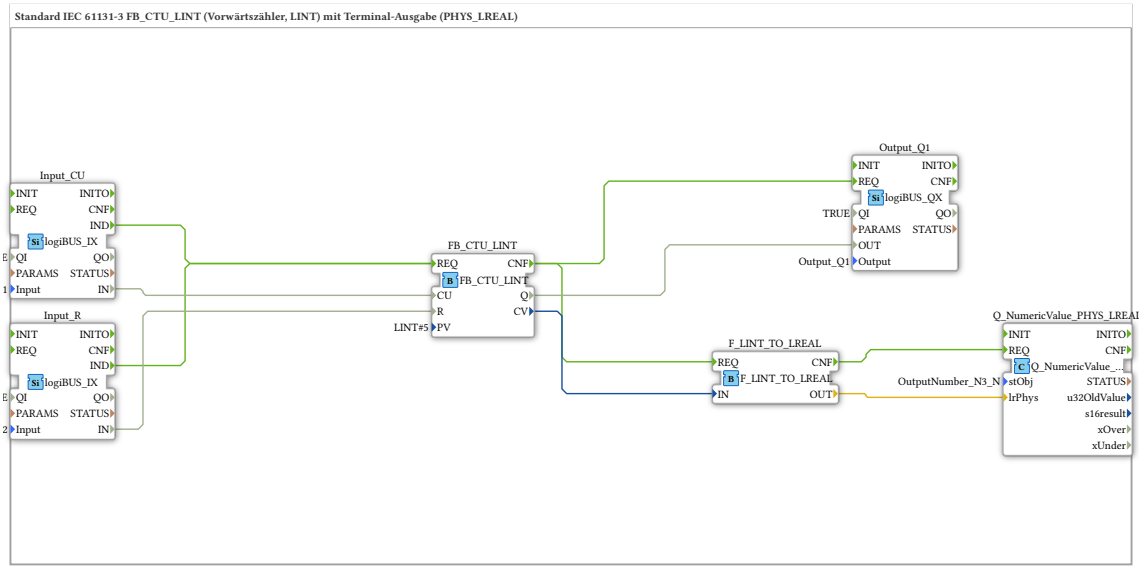
Zusammenfassung

Die Übung **Uebung_212** implementiert einen Vorwärtszähler (`FB_CTU_LINT`) mit einem Presetwert von 5. Über zwei digitale Eingänge wird gezählt und zurückgesetzt. Der Zählerstand wird mittels einer Datentypkonvertierung (`F_LINT_TO_UDINT`) auf einem Terminal ausgegeben, während der Ausgang `Output_Q1` aktiv wird, sobald der Zähler den Grenzwert erreicht. Die gezeigte Schaltung demonstriert die Interaktion zwischen industriellen Zählbausteinen, Ein-/Ausgabe und Benutzeranzeige – eine typische Aufgabenstellung für Automatisierungslösungen. Der beigefügte Kommentar zur Konvertierung gibt einen Hinweis auf mögliche Verbesserungen (Reduzierung der Ereignisse, Vermeidung negativer Zahlen).

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

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

2.2.1.379 Uebung_212b: Standard IEC 61131-3 FB_CTU_LINT (Vorwärtszähler, LINT) mit Terminal-Ausgabe (PHYS_LREAL)



Kein Bild verfügbar.

• – ○ • – ○ • – ○ •

Einleitung

Diese Übung implementiert einen Vorwärtszähler nach IEC 61131-3 (FB_CTU_LINT) mit Terminalausgabe des aktuellen Zählerwerts. Der Zähler wird durch einen digitalen Eingang (CU) hochgezählt und über einen weiteren digitalen Eingang (R) zurückgesetzt. Der Zählerstand wird nach jedem Zählereignis über eine Typkonvertierung (LINT ⟶ LREAL) auf einem Terminalbaustein (LogiBUS Utility) ausgegeben. Gleichzeitig wird ein digitaler Ausgang gesetzt, sobald der Zähler den voreingestellten Wert (PV = 5) erreicht hat.

Verwendete Funktionsbausteine (FBs)

- **FB_CTU_LINT**: Typ `iec61131::counters::FB_CTU_LINT`
  - Parameter: PV = LINT#5 (Voreinstellwert)
  - Ereigniseingang: REQ (Start der Zähleroperation)
  - Daten-Eingänge: CU (Count Up), R (Reset)
  - Daten-Ausgänge: Q (Erreicht Zählerstand ≥ PV), CV (aktueller Zählerwert)
- **Input_CU**: Typ `logiBUS::io::DI::logiBUS_IX`
  - Parameter: QI = TRUE, Input = Input_I1 (physischer Digitaleingang)
  - Ausgang: IN (Boolescher Wert des Eingangs)
  - Ereignisausgang: IND (Signalwechsel erkannt)
- **Input_R**: Typ `logiBUS::io::DI::logiBUS_IX`
  - Parameter: QI = TRUE, Input = Input_I2 (physischer Digitaleingang)
  - Ausgang: IN (Boolescher Wert des Eingangs)
  - Ereignisausgang: IND (Signalwechsel erkannt)
- **Output_Q1**: Typ `logiBUS::io::DQ::logiBUS_QX`
  - Parameter: QI = TRUE, Output = Output_Q1 (physischer Digitalausgang)
  - Dateneingang: OUT (Wert, der an den Ausgang übergeben wird)
- **F_LINT_TO_LREAL**: Typ `iec61131::conversion::F_LINT_TO_LREAL`
  - Dateneingang: IN (LINT-Wert)

  - Datenausgang: OUT (LREAL-Wert durch Konvertierung)
- **Q_NumericValue_PHYS_LREAL**: Typ `isobus::UT::Q::Q_NumericValue_PHYS_LREAL`
  - Parameter: stObj = OutputNumber_N3 (Verweis auf ein Terminal-Ausgabeobjekt)
  - Dateneingang: lrPhys (physikalischer LREAL-Wert)
  - Ereigniseingang: REQ (Auslösen der Terminal-Ausgabe)

Programmablauf und Verbindungen

Die Steuerung erfolgt über Ereignis- und Datenverbindungen:

Ereignisverbindungen

- `Input_CU.IND` ⟶ `FB_CTU_LINT.REQ` Bei einer positiven Flanke des Digitaleingangs I1 wird der Zähler zum Hochzählen angestoßen.
- `Input_R.IND` ⟶ `FB_CTU_LINT.REQ` Bei einer positiven Flanke des Digitaleingangs I2 wird der Zähler zurückgesetzt.
- `FB_CTU_LINT.CNF` ⟶ `Output_Q1.REQ` Nach Abschluss einer Zähleroperation (egal ob Zählen oder Reset) wird der Ausgang aktualisiert.
- `FB_CTU_LINT.CNF` ⟶ `F_LINT_TO_LREAL.REQ` Gleichzeitig wird die Typkonvertierung des aktuellen Zählerwerts angestoßen.
- `F_LINT_TO_LREAL.CNF` ⟶ `Q_NumericValue_PHYS_LREAL.REQ` Nach der Konvertierung wird der Wert an das Terminal gesendet.

Datenverbindungen

- `Input_CU.IN` ⟶ `FB_CTU_LINT.CU` Der Zustand von Eingang I1 steuert den Zählimpuls (positive Flanke).
- `Input_R.IN` ⟶ `FB_CTU_LINT.R` Der Zustand von Eingang I2 steuert den Reset (positive Flanke).
- `FB_CTU_LINT.Q` ⟶ `Output_Q1.OUT` Der Ausgang Q des Zählers (TRUE wenn CV >= PV) wird an den Digitalausgang Q1 weitergegeben.
- `FB_CTU_LINT.CV` ⟶ `F_LINT_TO_LREAL.IN` Der aktuelle Zählerwert (LINT) gelangt in den Konverter.
- `F_LINT_TO_LREAL.OUT` ⟶ `Q_NumericValue_PHYS_LREAL.lrPhys` Der konvertierte LREAL-Wert wird dem Terminal-Baustein übergeben.

Funktionsweise

1. Solange der Eingang CU (I1) einen positiven Flankenwechsel zeigt, erhöht der Zähler CV um 1.
2. Ein positiver Flankenwechsel am Reset-Eingang R (I2) setzt CV auf 0.
3. Erreicht der Zählerstand den voreingestellten Wert PV (hier 5) oder mehr, wird der Ausgang Q auf TRUE gesetzt. Ein weiteres Zählen ist dann nicht mehr möglich, bis ein Reset erfolgt.
4. Nach jedem Zähl- oder Reset-Vorgang wird der aktuelle CV-Wert auf dem Terminal (LogiBUS Utility) in physikalischer LREAL-Darstellung ausgegeben.

Lernziele:

- Verwendung eines IEC 61131-3 Zählers (FB_CTU_LINT)
- Parametrierung von Voreinstellwerten
- Ereignis- und Datenfluss zwischen Funktionsbausteinen
- Typkonvertierung (LINT ⟶ LREAL) und Terminalausgabe

Schwierigkeitsgrad: Fortgeschrittene Grundlagen **Vorkenntnisse:** Grundverständnis der 4diac-IDE, Umgang mit digitalen Ein-/Ausgängen, Ereignisverkettung.

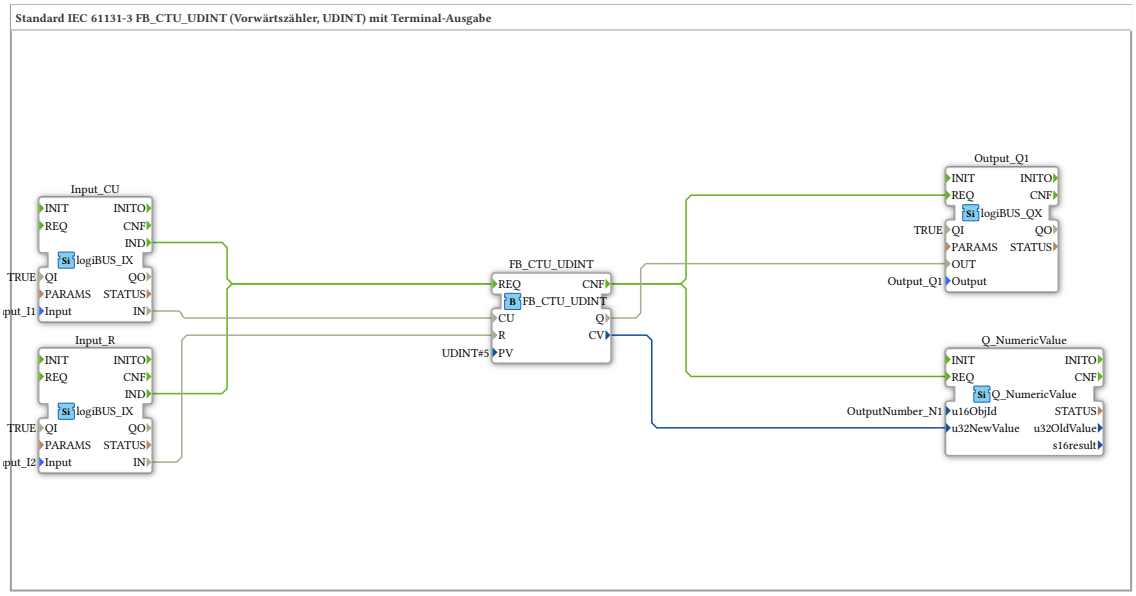
Zusammenfassung

Die Übung 212b zeigt die Kombination eines IEC 61131-3 Vorwärtszählers (FB_CTU_LINT) mit einer Terminalausgabe. Über zwei Digitaleingänge wird der Zähler gesteuert; der Ausgang Q schaltet bei Erreichen des Voreinstellwerts. Der aktuelle Zählerwert wird nach jeder Aktion über eine Typkonvertierung auf einem LogiBUS-Terminalbaustein ausgegeben. Dadurch wird die Verkettung von Ereignissen, die Datenflusslogik und die Nutzung von Utility-Bausteinen in 4diac demonstriert.

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

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

2.2.1.380 Uebung_213: Standard IEC 61131-3 FB_CTU_UDINT (Vorwärtszähler, UDINT) mit Terminal-Ausgabe



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung implementiert einen Vorwärtszähler nach IEC 61131-3 (Typ FB_CTU_UDINT) als Subapplikation. Der Zähler verfügt über zwei digitale Eingänge (Count-Up und Reset), einen digitalen Ausgang (Q) und eine Terminalausgabe für den aktuellen Zählwert. Die Eingänge werden über logiBUS-IO-Bausteine bereitgestellt, während der Ausgang und der numerische Wert auf konfigurierte logiBUS-Kanäle ausgegeben werden.

Verwendete Funktionsbausteine (FBs)

- **FB_CTU_UDINT** (Typ: `iec61131::counters::FB_CTU_UDINT`)
  - Parameter: `PV` = UDINT#5 (Vorgabewert für die Zählschwelle)
  - Ereignisse: REQ (Eingang), CNF (Ausgang)
  - Daten: CU (Count-Up), R (Reset), Q (Ausgang), CV (aktueller Zählwert)
- **Input_CU** (Typ: `logiBUS::io::DI::logiBUS_IX`)
  - Parameter: `QI` = TRUE, `Input` = Input_I1
  - Ereignis: IND (Ausgang)
  - Daten: IN (Ausgang)
- **Input_R** (Typ: `logiBUS::io::DI::logiBUS_IX`)
  - Parameter: `QI` = TRUE, `Input` = Input_I2
  - Ereignis: IND (Ausgang)
  - Daten: IN (Ausgang)
- **Output_Q1** (Typ: `logiBUS::io::DQ::logiBUS_QX`)
  - Parameter: `QI` = TRUE, `Output` = Output_Q1
  - Ereignis: REQ (Eingang)
  - Daten: OUT (Eingang)
- **Q_NumericValue** (Typ: `isobus::UT::Q::Q_NumericValue`)
  - Parameter: `u16ObjId` = OutputNumber_N1
  - Ereignis: REQ (Eingang)

– Daten: u32NewValue (Eingang)

Programmablauf und Verbindungen

Die Subapplikation besteht aus einer direkten Verschaltung der genannten Funktionsbausteine ohne zusätzliche Unterbausteine. Der Ablauf ist wie folgt:

1. **Eingangssignale**: Eine steigende Flanke am digitalen Eingang Input_I1 wird vom Baustein `Input_CU` erkannt und löst das Ereignis `IND` aus. Gleichzeitig wird der Wert des Eingangs (Bit) an den Datenausgang `IN` übergeben. Analog verhält es sich für den Reset-Eingang Input_I2 und den Baustein `Input_R`.
2. **Zählersteuerung**: Das Ereignis `IND` von `Input_CU` wird mit dem Ereigniseingang `REQ` des Zählers `FB_CTU_UDINT` verbunden. Über die Datenverbindung wird `Input_CU.IN` an den Zählereingang `CU` (Count-Up) gelegt. Dadurch wird der Zähler bei jeder positiven Flanke an diesem Eingang inkrementiert. Das Ereignis von `Input_R` wird ebenfalls auf den `REQ`-Eingang des Zählers geführt und der Datenwert an den `R`-Eingang (Reset) angelegt. Ein Reset setzt den Zähler auf Null.
3. **Zählerverhalten**: Der Zähler zählt ab dem Wert `0` aufwärts. Erreicht der interne Zählwert den Parameter `PV` (hier 5), wird der Ausgang `Q` auf TRUE gesetzt. Der aktuelle Zählwert ist am Ausgang `CV` (Datentyp UDINT) verfügbar.
4. **Ausgabe**: Nach jeder Zählerverarbeitung gibt der Zähler ein Bestätigungsereignis `CNF` aus. Dieses Ereignis wird parallel auf zwei Ausgabebausteine geschaltet:
   - **Output_Q1**: Das Ereignis `REQ` dieses Bausteins löst die Übernahme des Datenwertes von `FB_CTU_UDINT.Q` auf den physikalischen Ausgang `Output_Q1` aus.
   - **Q_NumericValue**: Das Ereignis `REQ` dieses Bausteins übernimmt den aktuellen Zählwert `CV` (als 32-Bit-Wert) und gibt ihn über die konfigurierte Objekt-ID `OutputNumber_N1` auf einem Terminal oder Display aus.

Ein Kommentar im Netzwerk weist darauf hin, dass eine zusätzliche Ereignisreduzierung (z.B. durch einen E_D_FF) zwischengeschaltet werden könnte, um die Ausgabe nur bei bestimmten Ereignissen zu aktualisieren.

Lernziele:

- Verständnis der IEC 61131-3 Zählerfunktionsbausteine.
- Zusammenspiel von Ereignis- und Datenflüssen in 4diac.
- Einbindung von digitalen Ein-/Ausgängen über logiBUS-IO-Bausteine.
- Ausgabe numerischer Werte auf ein Terminal mittels Q_NumericValue.

Benötigte Vorkenntnisse: Grundlegende Bedienung der 4diac-IDE, Verständnis von Ereignis-/Datenverbindungen, Kenntnis der logiBUS-IO-Konfiguration.

Übung starten: Die Subapplikation muss in ein 4diac-Projekt importiert werden. Anschließend sind die Hardware-Kanäle (Input_I1, Input_I2, Output_Q1, OutputNumber_N1) den tatsächlichen Ein-/Ausgängen der SPS (z.B. logiBUS) zuzuordnen. Nach dem Laden und Starten der Applikation können die Eingänge über Taster oder Simulationssignale getestet werden; der Zählwert und der Ausgangsstatus werden auf dem Terminal bzw. an den konfigurierten Ausgang ausgegeben.

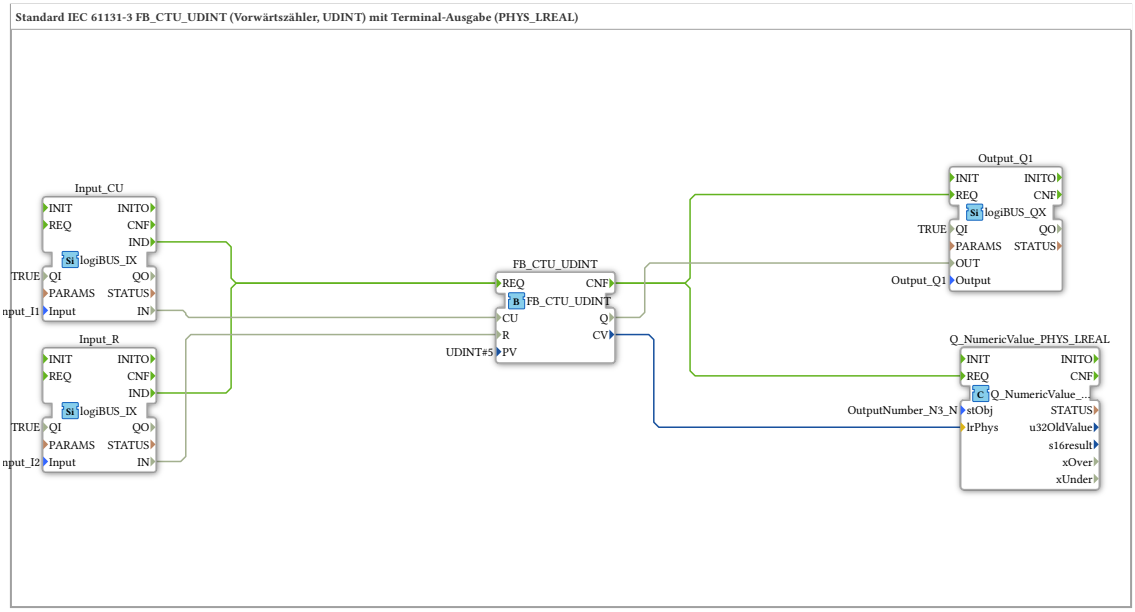
Zusammenfassung

Die Übung demonstriert die Nachbildung eines standardisierten IEC 61131-3 Vorwärtszählers (FB_CTU_UDINT) in 4diac. Durch die Verknüpfung von logiBUS-IO-Bausteinen mit einem Zähler und einer numerischen Ausgabe wird ein praxisnahes Beispiel für ereignisgesteuerte Automatisierungslogik gezeigt. Die Schaltung verdeutlicht, wie sowohl digitale als auch numerische Ausgaben parallel zu einer Zähleraktion erfolgen können. Dieser Aufbau eignet sich gut, um das Zusammenspiel von Ereignis- und Datenverbindungen in der 4diac-IDE zu erlernen und zu vertiefen.

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

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

2.2.1.381 Uebung_213b: Standard IEC 61131-3 FB_CTU_UDINT (Vorwärtszähler, UDINT) mit Terminal-Ausgabe (PHYS_LREAL)



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung implementiert einen Vorwärtszähler nach IEC 61131-3 (FB_CTU_UDINT) mit einem Presetwert von 5. Die Zählimpulse werden über zwei digitale Eingänge bereitgestellt:

- **I1** dient als Zähleingang (CU – Count Up)
- **I2** dient als Rücksetzeingang (R – Reset)

Der Zählerstand (CV) wird als physikalische Größe vom Typ LREAL auf einem Terminal ausgegeben. Ein digitaler Ausgang (Q1) wird aktiviert, sobald der Zählerstand den Presetwert erreicht oder überschreitet.

Die Übung zeigt die direkte Verbindung eines UDINT-Werts auf einen LREAL-Ausgang – eine Typumwandlung ist nicht erforderlich, da UDINT implizit nach LREAL konvertiert werden kann.

Verwendete Funktionsbausteine (FBs)

Die Übung enthält fünf Funktionsbausteine, die alle auf der obersten Netzwerkebene angeordnet sind. Sub-Bausteine sind nicht vorhanden.

FB_CTU_UDINT

- **Typ**: `iec61131::counters::FB_CTU_UDINT`
- **Parameter**: `PV` = `UDINT#5` (Presetwert)
- **Funktionsweise**: Vorwärtszähler (IEC 61131-3) für ganzzahlige Werte vom Typ UDINT. Bei jedem steigenden Signal am Eingang CU wird der Zählerstand CV um 1 erhöht. Wird der Eingang R gesetzt, wird CV auf 0 zurückgesetzt. Der Ausgang Q wird TRUE, sobald CV ≥ PV.

Input_CU

- **Typ**: `logiBUS::io::DI::logiBUS_IX`
- **Parameter**: `QI` = `TRUE`, `Input` = `Input_I1`
- **Funktionsweise**: Digitaler Eingangsbaustein, der den physischen Eingang I1 (z. B. Taster) ausliest. Der Ereignisausgang IND signalisiert eine Zustandsänderung.

Input_R

- **Typ**: `logiBUS::io::DI::logiBUS_IX`
- **Parameter**: `QI` = `TRUE`, `Input` = `Input_I2`
- **Funktionsweise**: Digitaler Eingangsbaustein für den physischen Eingang I2 (Rücksetztaster). Entspricht im Aufbau dem Baustein Input_CU.

Output_Q1

- **Typ**: `logiBUS::io::DQ::logiBUS_QX`
- **Parameter**: `QI` = `TRUE`, `Output` = `Output_Q1`
- **Funktionsweise**: Digitaler Ausgangsbaustein, der den physischen Ausgang Q1 (z. B. Lampe) schaltet. Der Wert des Datenports OUT wird direkt an die Hardware übergeben.

Q_NumericValue_PHYS_LREAL

- **Typ**: `isobus::UT::Q::Q_NumericValue_PHYS_LREAL`
- **Parameter**: `stObj` = `OutputNumber_N3`
- **Funktionsweise**: Terminalausgabe für einen physikalischen LREAL-Wert. Der übergebene Wert (lrPhys) wird auf dem konfigurierten Ausgabekanal (hier `OutputNumber_N3`) dargestellt.

Programmablauf und Verbindungen

Der Ablauf wird durch **Ereignisverbindungen** gesteuert:

1. Eine Änderung an **Input_CU** (Taster I1) erzeugt das Ereignis **IND**, das den Zählerbaustein über seinen **REQ**-Eingang triggert.
2. Ebenso erzeugt eine Änderung an **Input_R** (Taster I2) ein **IND**-Ereignis, das ebenfalls den **REQ**-Eingang des Zählers erreicht – ein und derselbe Zähler wird also von beiden Eingängen angesprochen.
3. Nach der Verarbeitung gibt der Zähler ein **CNF**-Ereignis aus. Dieses wird parallel an zwei Bausteine weitergeleitet:
   - an **Output_Q1** (Ausgang Q1)
   - an **Q_NumericValue_PHYS_LREAL** (Terminalausgabe)

Die **Datenverbindungen** übertragen die Werte:

- `Input_CU.IN` ⟶ `FB_CTU_UDINT.CU` (Zählimpuls)
- `Input_R.IN` ⟶ `FB_CTU_UDINT.R` (Rücksetzsignal)
- `FB_CTU_UDINT.Q` ⟶ `Output_Q1.OUT` (Ausgangszustand)
- `FB_CTU_UDINT.CV` ⟶ `Q_NumericValue_PHYS_LREAL.lrPhys` (Zählerstand als LREAL)

Hinweis: Der Zählerstand CV ist vom Typ UDINT. Der Datenport lrPhys erwartet LREAL. Da in IEC 61131-3 UDINT ohne explizite Konvertierung auf LREAL geschlossen werden kann, ist diese direkte Verbindung zulässig.

Zusammenfassung

Die Übung veranschaulicht den Einsatz eines standardisierten Vorwärtszählers (FB_CTU_UDINT) in der 4diac-IDE. Sie zeigt:

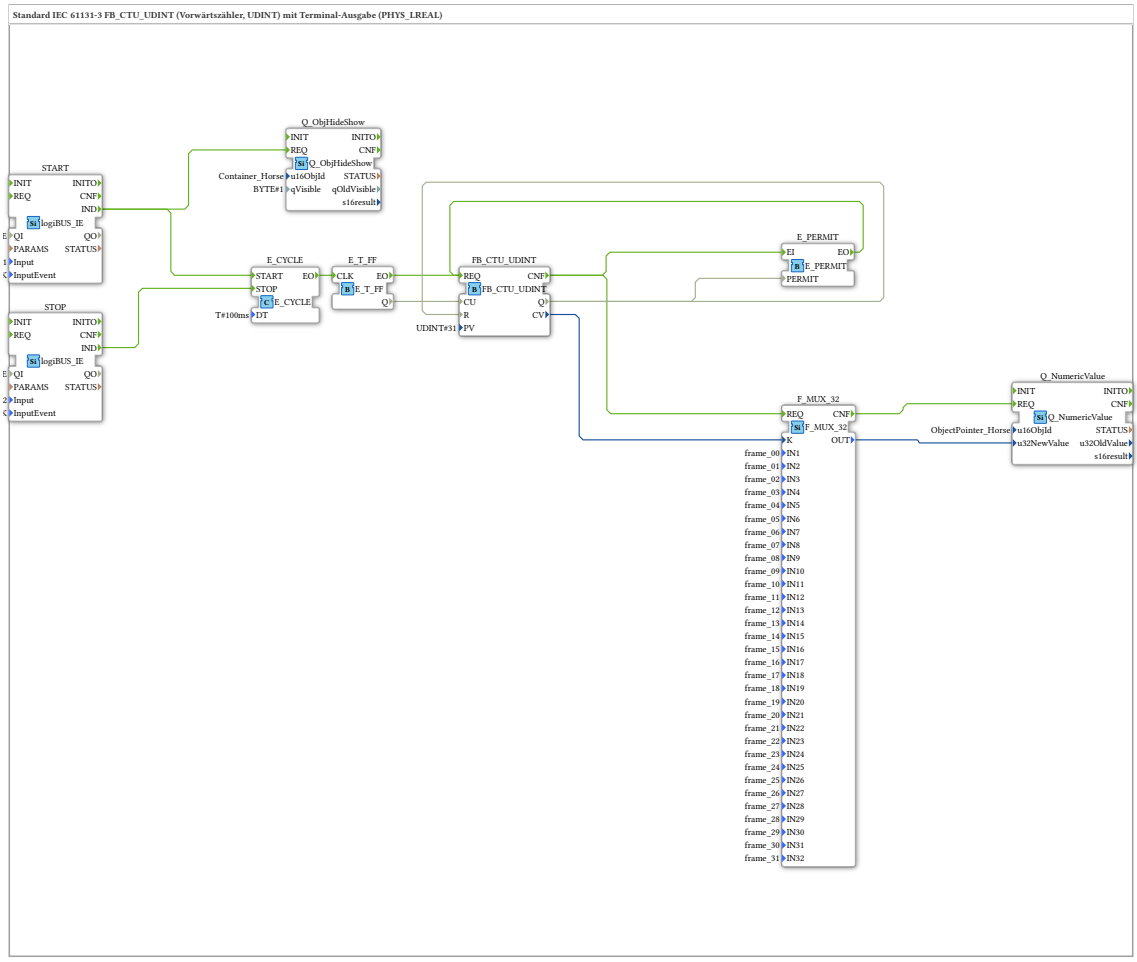
- Anbindung zweier digitaler Eingänge als Zähl- und Rücksetzsignal
- Verwendung eines digitalen Ausgangs zur Anzeige des Erreichens des Presetwerts
- Ausgabe des Zählerstands auf einem Terminal mithilfe eines physikalischen LREAL-Ausgabebausteins
- implizite Typkonvertierung von UDINT nach LREAL

Damit werden grundlegende Kenntnisse der IEC-61131-3-Zählerfunktionen und der Ein-/Ausgangsverknüpfung in 4diac vermittelt.

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

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

2.2.1.382 Uebung_213c: Standard IEC 61131-3 FB_CTU_UDINT (Vorwärtszähler, UDINT) mit Terminal-Ausgabe (PHYS_LREAL)



• — ○ • — ○ • — ○ •

Einleitung

Diese Übung realisiert einen Vorwärtszähler nach IEC 61131-3 (FB_CTU_UDINT) mit einer Zählgrenze von 31. Der Zählerstand wird zyklisch aktualisiert und über einen Multiplexer auf eine numerische Terminalausgabe (PHYS_LREAL) übertragen. Zusätzlich wird ein animiertes Objekt („Pferd“) über Ein-/Ausblenden gesteuert. Die Übung veranschaulicht die Kombination von IEC 61131-3 Funktionsbausteinen mit ereignisgesteuerter 4diac-Logik und Terminalausgabe.

Verwendete Funktionsbausteine (FBs)

Sub-Bausteine: `FB_CTU_UDINT`

- **Typ**: `iec61131::counters::FB_CTU_UDINT`
- **Verwendete interne FBs**: (Standard IEC 61131-3, keine weiteren Unterbausteine)
- **Parameter**:
  - `PV` = `UDINT#31` (Zählgrenze)
- **Funktionsweise**: Der Baustein zählt bei jedem positiven Flanke am Eingang `CU` den aktuellen Zählerstand `CV` (UDINT) hoch. Erreicht `CV` den Wert `PV`, wird der Ausgang `Q` gesetzt. Der Zähler kann über den Eingang `R` zurückgesetzt werden.

Sub-Bausteine: `START`

- **Typ**: `logiBUS::io::DI::logiBUS_IE`
- **Parameter**:
  - `QI` = `TRUE`
  - `Input` = `Input_I1` (physischer Eingang I1)

  - `InputEvent` = `BUTTON_SINGLE_CLICK`
- **Funktionsweise**: Erzeugt bei Betätigung des Tasters I1 (Einfachklick) ein Ereignis `IND`, das den Zyklusstart und das Einblenden des Objekts auslöst.

Sub-Bausteine: `STOP`

- **Typ**: `logiBUS::io::DI::logiBUS_IE`
- **Parameter**:
  - `QI` = `TRUE`
  - `Input` = `Input_I2`
  - `InputEvent` = `BUTTON_SINGLE_CLICK`
- **Funktionsweise**: Erzeugt bei Betätigung des Tasters I2 ein Ereignis `IND`, das den zyklischen Timer anhält.

Sub-Bausteine: `E_CYCLE`

- **Typ**: `iec61499::events::E_CYCLE`
- **Parameter**:
  - `DT` = `T#100ms` (Zykluszeit 100 ms)
- **Funktionsweise**: Ein zyklischer Ereignisgenerator. Mit `START` wird der Zyklus gestartet, mit `STOP` angehalten. Das Ausgangsereignis `EO` tritt alle 100 ms auf.

Sub-Bausteine: `E_T_FF`

- **Typ**: `iec61499::events::E_T_FF`
- **Funktionsweise**: Ein T-Flipflop (Toggle-Flipflop). Jedes Ereignis am Takteingang `CLK` ändert den Zustand des Ausgangs `Q`. Hier wird aus dem 100-ms-Takt ein 200-ms-Takt (wenn Q=1) erzeugt, um den Zählimpuls `CU` zu generieren.

Sub-Bausteine: `E_PERMIT`

- **Typ**: `iec61499::events::E_PERMIT`
- **Funktionsweise**: Ein Freigabebaustein. Er leitet ein Ereignis von `EI` nach `EO` nur dann weiter, wenn der Eingang `PERMIT` `TRUE` ist. Hier wird die Datenausgabe nur freigegeben, wenn der Zähler seinen Endwert erreicht hat (`Q=1`).

Sub-Bausteine: `F_MUX_32`

- **Typ**: `iec61131::selection::F_MUX_32`
- **Parameter**:
  - `IN1` … `IN32` = `frame_00` … `frame_31` (32 vordefinierte Konstanten)
- **Funktionsweise**: Ein 32-Kanal-Multiplexer. Der Ausgang `OUT` entspricht dem Eingang `IN(K)`, wobei `K` der Auswahlwert (UDINT) ist. Hier wird der aktuelle Zählerstand `CV` als Auswahl verwendet, um das entsprechende Frame für die Animation auszuwählen.

Sub-Bausteine: `Q_NumericValue`

- **Typ**: `isobus::UT::Q::Q_NumericValue`
- **Parameter**:
  - `u16ObjId` = `ObjectPointer_Horse`
- **Funktionsweise**: Schreibt den am Eingang `u32NewValue` anliegenden Wert in ein Terminal-Display-Objekt. Der Wert wird als physikalische LREAL-Größe dargestellt.

Sub-Bausteine: `Q_ObjHideShow`

- **Typ**: `isobus::UT::Q::Q_ObjHideShow`
- **Parameter**:
  - `u16ObjId` = `Container_Horse`
  - `qVisible` = `BYTE#1` (sichtbar)
- **Funktionsweise**: Blendet ein Grafik-Container-Objekt ein (bei `REQ` wird es sichtbar). Dieses Objekt enthält vermutlich die animierte Pferdegrafik.

Programmablauf und Verbindungen

1. **Start**: Ein Druck auf Taster **I1** erzeugt das Ereignis `START.IND`. Dieses startet den zyklischen Timer `E_CYCLE` und blendet gleichzeitig das Objekt `Container_Horse` über `Q_ObjHideShow` ein.

2. **Zyklischer Takt**: `E_CYCLE` erzeugt alle 100 ms ein Ereignis `EO`. Dieses taktet das T-Flipflop `E_T_FF`, dessen Ausgang `Q` wechselt bei jedem zweiten Ereignis seinen Zustand. Dadurch entsteht ein 200-ms-Takt am Ausgang des Flipflops.
3. **Zählung**: Der Ausgang `E_T_FF.Q` ist mit dem Zähleingang `CU` von `FB_CTU_UDINT` verbunden. Bei jeder positiven Flanke (Wechsel von 0 auf 1) erhöht der Zähler `CV` um 1.
4. **Rücksetzen**: Sobald der Zähler seinen Endwert 31 erreicht hat, wird `Q` gesetzt. Dieser Zustand wird auf den Rücksetzeingang `R` zurückgeführt **und** an den Freigabeeingang `E_PERMIT.PERMIT` gelegt. Dadurch wird der Zähler automatisch zurückgesetzt und gleichzeitig die weitere Verarbeitung freigegeben.
5. **Datenauswahl**: Der aktuelle Zählerstand `CV` (vor dem Rücksetzen) wird als Auswahl `K` an den Multiplexer `F_MUX_32` gelegt. Der Multiplexer wählt das entsprechende `frame_xx` aus (0 … 31).
6. **Terminalausgabe**: Das vom Multiplexer gelieferte `frame_xx` wird an den Numeric-Wert-Ausgabebaustein `Q_NumericValue` übergeben und auf dem angeschlossenen Terminal dargestellt (z. B. als Zeichen oder Grafik).
7. **Stopp**: Ein Druck auf Taster **I2** erzeugt `STOP.IND`, das den Timer `E_CYCLE` anhält. Zählung und Ausgabe werden gestoppt.

Lernziele:

- Anwendung des IEC 61131-3 Zählers `FB_CTU_UDINT` in einer 4diac-Umgebung.
- Kombination von ereignisgesteuerten Bausteinen (E_CYCLE, E_T_FF, E_PERMIT) mit datenflussorientierten Bausteinen (F_MUX_32, Q_NumericValue).
- Steuerung eines animierten Objekts über Taster, zyklischen Timer und Zähler.

Schwierigkeitsgrad: Mittel **Vorkenntnisse**: Grundlagen der IEC 61499 Ereignissteuerung, IEC 61131-3 Zählerfunktionen.

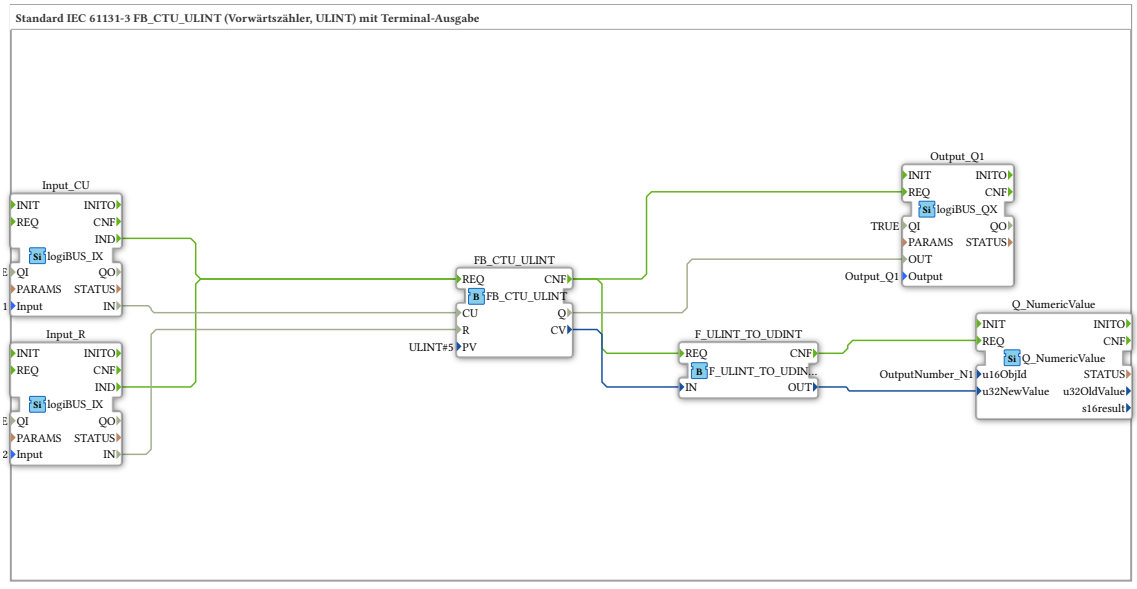
Zusammenfassung

Die Übung `Uebung_213c` führt einen Vorwärtszähler mit automatischem Reset bei Erreichen der Grenze 31 aus. Ein zyklischer Timer (100 ms) erzeugt über ein T-Flipflop einen 200-ms-Takt für die Zählimpulse. Der Zählerstand wird über einen Multiplexer in ein numerisches Terminalformat umgesetzt und angezeigt. Zusätzlich steuern zwei Taster den Start/Stopp der gesamten Animation. Die Übung zeigt anschaulich die Integration von IEC 61131-3 Zählerlogik in ein ereignisgesteuertes 4diac-System mit grafischer Ausgabe.

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

- 🌐 Eclipse 4diac IDE & Farb-Referenz auf ms-muc-docs.de
- 🌐 IEC 61499 Events – Der Puls der Automatisierung auf ms-muc-docs.de

2.2.1.383 Uebung_214: Standard IEC 61131-3 FB_CTU_ULINT (Vorwärtszähler, ULINT) mit Terminal-Ausgabe



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert die Verwendung des IEC 61131-3 Standardvorwärtszählers **FB_CTU_ULINT** für den Datentyp **ULINT** (unsigned long integer). Der Zähler wird über zwei digitale Eingänge gesteuert: Der Eingang **CU** zählt bei jeder positiven Flanke um eins hoch, während der Eingang **R** den Zählerstand zurücksetzt. Der aktuelle Zählerstand wird auf ein Terminal ausgegeben, und der Ausgang **Q** wird aktiv, sobald der Zählerstand den voreingestellten Wert **PV** erreicht oder überschreitet. Ziel ist es, die Integration eines Standard-FBs mit Hardware-Ein-/Ausgängen und einer textuellen Ausgabe zu verstehen.

Verwendete Funktionsbausteine (FBs)

Die Übung besteht aus einem Netzwerk von sechs internen Funktionsbausteinen, die über Ereignis- und Datenverbindungen verknüpft sind.

Sub-Bausteine: FB_CTU_ULINT

- **Typ**: `iec61131::counters::FB_CTU_ULINT`
- **Verwendete interne FBs**: keiner (der Baustein selbst ist primitiv)
- **Parameter**:
  - `PV = ULINT#5` – Der Zähler wird aktiv, sobald der Stand den Wert 5 erreicht.
- **Ereignisseingänge/-ausgänge**:
  - **REQ** (Eingang) – löst die Zählerlogik aus.
  - **CNF** (Ausgang) – signalisiert die Verarbeitung und gibt das Ergebnis weiter.
- **Dateneingänge/-ausgänge**:
  - **CU** (Eingang, BOOL) – Zählimpuls (steigende Flanke zählt).
  - **R** (Eingang, BOOL) – Rücksetzen des Zählerstands.
  - **PV** (Eingang, ULINT) – Voreinstellwert, hier 5.
  - **Q** (Ausgang, BOOL) – TRUE, wenn CV >= PV.
  - **CV** (Ausgang, ULINT) – aktueller Zählerstand.
- **Funktionsweise**: Bei jedem REQ-Ereignis wird die Zählerlogik ausgeführt: Eine steigende Flanke auf **CU** erhöht **CV** um 1, ein TRUE auf **R** setzt **CV** auf 0. **Q** wird in derselben Verarbeitung aktualisiert.

Sub-Bausteine: Input_CU und Input_R (logiBUS_IX)

- **Typ**: `logiBUS::io::DI::logiBUS_IX`
- **Verwendete interne FBs**: keiner
- **Parameter**:
  - `QI = TRUE` – Qualifier aktiviert den Kanal.
  - `Input = Input_I1` (für Input_CU) bzw. `Input = Input_I2` (für Input_R) – Hardware-Pin Zuordnung.
- **Ereignisse**:
  - **IND** (Ausgang) – wird bei Signaländerung am Eingang ausgelöst.
- **Daten**:
  - **IN** (Ausgang, BOOL) – aktueller Zustand des digitalen Eingangs.
- **Funktionsweise**: Diese Bausteine lesen die realen digitalen Eingänge (logiBUS-Hardware) und geben bei jeder Änderung ein Ereignis aus. Der Zustand wird über **IN** bereitgestellt.

Sub-Bausteine: Output_Q1 (logiBUS_QX)

- **Typ**: `logiBUS::io::DQ::logiBUS_QX`
- **Verwendete interne FBs**: keiner
- **Parameter**:
  - `QI = TRUE` – Qualifier aktiviert den Ausgang.
  - `Output = Output_Q1` – Hardware-Pin Zuordnung.
- **Ereignisse**:
  - **REQ** (Eingang) – löst die Ausgabe des anliegenden Werts aus.
- **Daten**:
  - **OUT** (Eingang, BOOL) – der auszugebende Wert.
- **Funktionsweise**: Der Baustein setzt den Digitalausgang auf den Wert von **OUT**, sobald ein **REQ**-Ereignis eintrifft.

Sub-Bausteine: F_ULINT_TO_UDINT

- **Typ**: `iec61131::conversion::F_ULINT_TO_UDINT`
- **Verwendete interne FBs**: keiner
- **Parameter**: keine

- **Ereignisse**:
  - **REQ** (Eingang) – startet die Konvertierung.
  - **CNF** (Ausgang) – signalisiert, dass das Ergebnis bereit ist.
- **Daten**:
  - **IN** (Eingang, ULINT) – der zu konvertierende Wert.
  - **OUT** (Ausgang, UDINT) – das konvertierte Ergebnis.
- **Funktionsweise**: Der Baustein wandelt einen 64-Bit unsigned Integer (ULINT) in einen 32-Bit unsigned Integer (UDINT) um. Ein Überlauf kann auftreten, wenn der ULINT-Wert größer als 2^{32}-1 ist (Hinweis im Netzwerk).

Sub-Bausteine: Q_NumericValue

- **Typ**: `isobus::UT::Q::Q_NumericValue`
- **Verwendete interne FBs**: keiner
- **Parameter**:
  - `u16ObjId = OutputNumber_N1` – Identifikation des Terminal-Objekts, an das der Wert gesendet wird.
- **Ereignisse**:
  - **REQ** (Eingang) – löst die Aktualisierung der Anzeige aus.
- **Daten**:
  - **u32NewValue** (Eingang, UDINT) – der anzuzeigende numerische Wert.
- **Funktionsweise**: Dieser Baustein sendet den übergebenen 32-Bit Wert an ein Terminal (z. B. HMI oder Konsole), sodass der aktuelle Zählerstand visuell ausgegeben wird.

Programmablauf und Verbindungen

1. **Eingangsereignisse**: Die digitalen Eingänge **Input_CU** und **Input_R** erzeugen bei Zustandsänderungen ein **IND**-Ereignis. Beide Ereignisse sind mit dem **REQ**-Eingang des Zählers **FB_CTU_ULINT** verbunden. Dadurch wird der Zähler jedes Mal neu berechnet, wenn einer der Eingänge seinen Zustand ändert.
2. **Zählerlogik**: Der Zähler wertet die anliegenden Daten aus:
   - **CU** erhält den aktuellen Zustand von **Input_CU.IN**.
   - **R** erhält den Zustand von **Input_R.IN**.
   - Eine steigende Flanke auf **CU** erhöht den internen Zählstand **CV** um 1.
   - Ein TRUE auf **R** setzt **CV** auf 0.
   - Überschreitet **CV** den Wert **PV** (hier 5), wird **Q** auf TRUE gesetzt.
3. **Ausgabe**: Nach der Berechnung sendet der Zähler das **CNF**-Ereignis. Dieses Ereignis wird parallel zu zwei Bausteinen geführt:
   - **Output_Q1** setzt den digitalen Ausgang auf den Wert von **FB_CTU_ULINT.Q**.
   - **F_ULINT_TO_UDINT** konvertiert den aktuellen Zählerstand **CV** von ULINT in UDINT.
4. **Terminal-Ausgabe**: Nach Abschluss der Konvertierung löst **F_ULINT_TO_UDINT.CNF** das **REQ**-Ereignis von **Q_NumericValue** aus. Der konvertierte Wert wird über die Datenleitung **OUT** an **u32NewValue** übergeben und auf dem Terminal ausgegeben.
5. **Hinweise aus der Übung**:
   - Ein Kommentar schlägt vor, ggf. einen **E_D_FF** (Einschaltverzögerer) einzubauen, um die Anzahl der Ereignisse zu reduzieren (z. B. bei schnellen Eingangssignalen).
   - Ein weiterer Kommentar warnt vor einem möglichen **Überlauf** bei der Konvertierung von ULINT nach UDINT, da ULINT einen größeren Wertebereich abdecken kann (bis 2^{64}-1) als UDINT (bis 2^{32}-1).

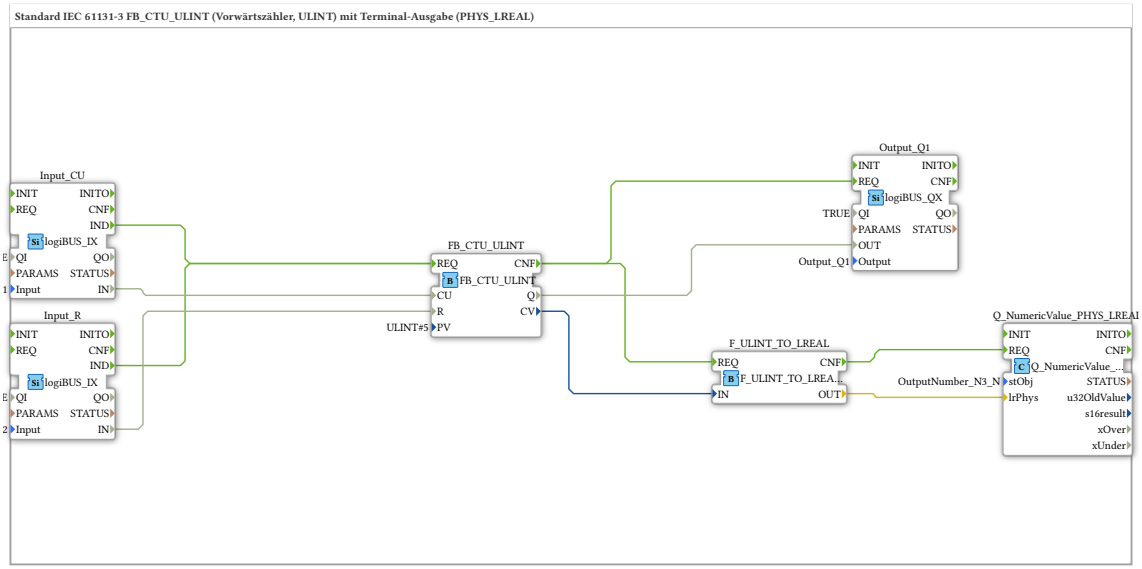
Zusammenfassung

Die Übung 214 zeigt den praktischen Einsatz des IEC 61131-3 Standard-Zählers **FB_CTU_ULINT** in einer 4diac-IDE Umgebung. Durch die Verknüpfung mit Hardware-Eingängen (logiBUS) und einem Terminalausgabebaustein wird deutlich, wie ein industrieller Zähler konfiguriert und visualisiert werden kann. Die Schüler lernen, Ereignis- und Datenflüsse zu verstehen, sowie die Bedeutung von Datentypkonvertierung und die Gefahr von Überläufen bei nicht-passenden Typen. Die Übung eignet sich für Fortgeschrittene, die bereits Grundlagen der IEC 61131-3 und der 4diac-IDE beherrschen.

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

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

2.2.1.384 Uebung_214b: Standard IEC 61131-3 FB_CTU_ULINT (Vorwärtszähler, ULINT) mit Terminal-Ausgabe (PHYS_LREAL)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung implementiert einen Vorwärtszähler nach IEC 61131-3 (FB_CTU_ULINT). Der Zähler erhöht seinen Stand bei jeder steigenden Flanke am Eingang CU (Count Up) um eins, sofern der Rücksetzeingang R nicht aktiv ist. Wird der Presetwert (PV) erreicht oder überschritten, setzt der Ausgang Q auf TRUE. Der aktuelle Zählerstand wird als ULINT (vorzeichenloser 64-Bit-Integer) ausgegeben, über einen Konvertierungsbaustein in LREAL umgewandelt und an einen Terminal-Ausgabebaustein übergeben, der den Wert auf einem angeschlossenen Terminal anzeigt.

Die physikalischen Ein- und Ausgänge sind mit den logiBUS-Klemmen Input_I1, Input_I2 und Output_Q1 verbunden.

Verwendete Funktionsbausteine (FBs)

FB_CTU_ULINT

- **Typ**: `iec61131::counters::FB_CTU_ULINT`
- **Parameter**:
  - `PV` = `ULINT#5` (Presetwert)
- **Ereigniseingang/-ausgang**:
  - `REQ` (Eingang) – Trigger für Zähl- oder Rücksetzoperation
  - `CNF` (Ausgang) – Bestätigung der Bearbeitung
- **Dateneingang/-ausgang**:
  - `CU` (Eingang) – Zählimpuls (steigende Flanke)
  - `R` (Eingang) – Rücksetzen des Zählers
  - `Q` (Ausgang) – TRUE wenn Zählerstand >= PV
  - `CV` (Ausgang) – aktueller Zählerstand (ULINT)

Input_CU (logiBUS_IX)

- **Typ**: `logiBUS::io::DI::logiBUS_IX`
- **Parameter**:
  - `QI` = `TRUE` (Aktivierung)
  - `Input` = `Input_I1` (physikalischer Digitaleingang)
- **Ereignisausgang**: `IND` – meldet ein Ereignis bei Änderung des Eingangs
- **Datenausgang**: `IN` – aktueller Zustand des Eingangs

Input_R (logiBUS_IX)

- **Typ**: `logiBUS::io::DI::logiBUS_IX`
- **Parameter**:
  - `QI` = `TRUE`
  - `Input` = `Input_I2`
- **Ereignisausgang**: `IND`
- **Datenausgang**: `IN`

Output_Q1 (logiBUS_QX)

- **Typ**: `logiBUS::io::DQ::logiBUS_QX`
- **Parameter**:
  - `QI` = `TRUE`
  - `Output` = `Output_Q1` (physikalischer Digitalausgang)
- **Ereigniseingang**: `REQ` – löst das Setzen des Ausgangs aus
- **Dateneingang**: `OUT` – Wert, der auf den Ausgang geschrieben wird

F_ULINT_TO_LREAL

- **Typ**: `iec61131::conversion::F_ULINT_TO_LREAL`
- **Ereigniseingang/-ausgang**:
  - `REQ` (Eingang) – startet Konvertierung
  - `CNF` (Ausgang) – Bestätigung der Konvertierung
- **Dateneingang/-ausgang**:
  - `IN` (Eingang) – ULINT-Wert
  - `OUT` (Ausgang) – konvertierter LREAL-Wert

Q_NumericValue_PHYS_LREAL

- **Typ**: `isobus::UT::Q::Q_NumericValue_PHYS_LREAL`
- **Parameter**:
  - `stObj` = `OutputNumber_N3` (Referenz auf das Terminal-Ausgabeobjekt)
- **Ereigniseingang**: `REQ` – löst Ausgabe aus
- **Dateneingang**: `lrPhys` – physikalischer Wert als LREAL

Programmablauf und Verbindungen

1. **Zählimpuls (CU)**: Eine steigende Flanke am Digitaleingang Input_I1 wird vom Baustein `Input_CU` erkannt und über den Ereignisausgang `IND` an den `REQ`-Eingang des Zählers `FB_CTU_ULINT` weitergeleitet. Gleichzeitig wird der Signalzustand über die Datenverbindung an den `CU`-Eingang übergeben.
2. **Rücksetzen (R)**: Ein Signal am Digitaleingang Input_I2 wird analog über `Input_R` an den `R`-Eingang des Zählers geführt. Bei aktivem Signal wird der Zähler auf 0 gesetzt.
3. **Zählerverarbeitung**: Der Zähler erhöht bei jeder positiven Flanke an `CU` den internen Stand, solange `R` = FALSE. Wird der Presetwert (PV = 5) erreicht, setzt der Ausgang `Q` auf TRUE.
4. **Ausgabe des Zählerstands (CV)**: Nach jedem Zähl- oder Rücksetzvorgang signalisiert `FB_CTU_ULINT` über `CNF` die Fertigstellung. Dieses Ereignis triggert gleichzeitig zwei Zweige:
   - **Digitalausgang**: Das Ereignis `CNF` startet den Baustein `Output_Q1`. Der Wert von `Q` (TRUE/FALSE) wird auf den physikalischen Ausgang Output_Q1 geschrieben.
   - **Terminalausgabe**: Ebenfalls über `CNF` wird der Konvertierungsbaustein `F_ULINT_TO_LREAL` getriggert. Dieser wandelt den aktuellen Zählerstand (`CV`, ULINT) in LREAL um. Nach Abschluss der Konvertierung wird das Terminal-Ausgabemodul `Q_NumericValue_PHYS_LREAL` aktiviert und der konvertierte Wert angezeigt.

Alle Datenverbindungen sind so verdrahtet, dass die Werte synchron mit den Ereignissen weitergegeben werden.

Lernziele:

- Verständnis des IEC 61131-3 Zählers FB_CTU_ULINT (Vorwärtszähler)
- Umgang mit Datentypkonvertierung (ULINT → LREAL)
- Einbindung von physikalischen Ein-/Ausgängen und Terminalausgabe in einer 4diac-Subapplikation

Schwierigkeitsgrad: Einfach **Benötigte Vorkenntnisse**: Grundlagen der 4diac-IDE, Umgang mit logiBUS-Klemmen

Ausführung: In der 4diac-IDE die Subapplikation `Uebung_214b` öffnen und im Simulation-Modus testen. Die Eingänge `Input_I1` und `Input_I2` können über die Hardware-Simulation oder reale Klemmen gesteuert werden.

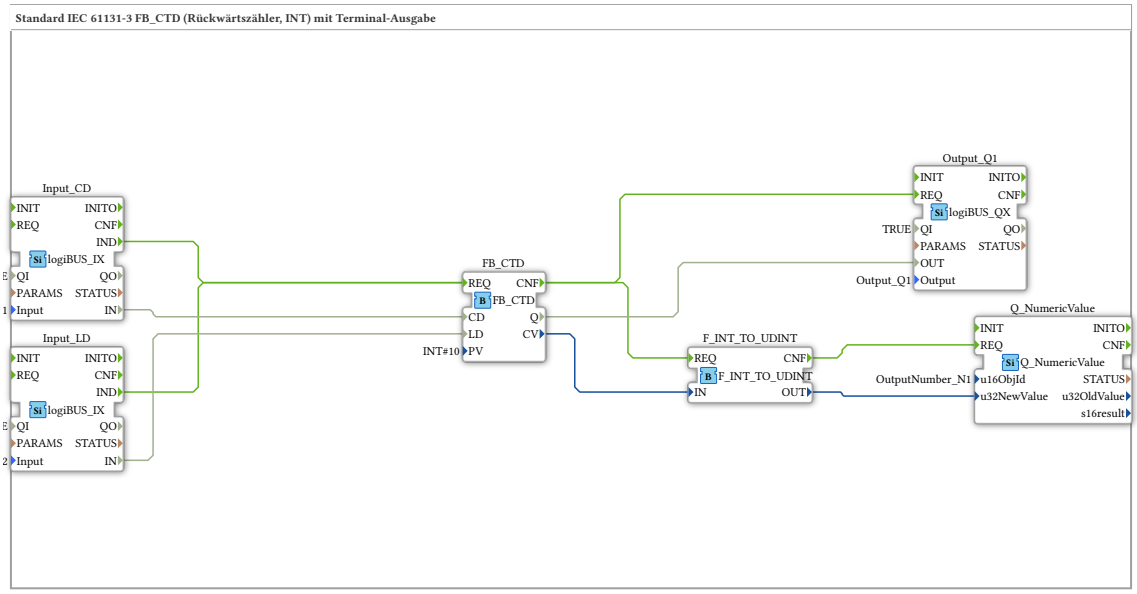
Zusammenfassung

Die Übung 214b demonstriert die Implementierung eines industriellen Vorwärtszählers mit Terminal-Ausgabe. Der Zähler wird über zwei digitale Eingänge gesteuert, sein Ausgang schaltet einen Digitalausgang, und der aktuelle Zählerstand wird nach Konvertierung auf einem Terminal ausgegeben. Das Zusammenspiel von Ereignis- und Datenverbindungen sowie die Verwendung von Standard-IEC- und logiBUS-Bausteinen bilden die Grundlage für komplexere Automatisierungsanwendungen.

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

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

2.2.1.385 Uebung_215: Standard IEC 61131-3 FB_CTD (Rückwärtszähler, INT) mit Terminal-Ausgabe



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung realisiert einen **Rückwärtszähler (FB_CTD)** gemäß IEC 61131-3. Der Zähler zählt bei jedem Ereignis an seinem **CD**-Eingang (Count Down) von einem vorgegebenen **PV**-Wert (Preset Value) herunter. Mit dem **LD**-Eingang (Load) kann der Zähler jederzeit auf den Preset-Wert zurückgesetzt werden. Der aktuelle Zählerstand wird auf einem numerischen Ausgabefeld (Terminal) dargestellt, und ein binärer Ausgang (Q1) wird gesetzt, sobald der Zählerstand **0** erreicht.

Die Übung bildet einen typischen Anwendungsfall eines IEC-Zählbausteins ab und zeigt dessen Anbindung an Hardwareeingänge sowie die Ausgabe des Zählerwertes über einen isobus-Terminalbaustein.

Verwendete Funktionsbausteine (FBs)

Die gesamte Schaltung besteht aus einem SubApp-Typ (SubAppType) mit der Bezeichnung „Uebung_215“. Im FBNetzwerk sind folgende Bausteine enthalten:

Sub-Bausteine

`FB_CTD` (Typ: `iec61131::counters::FB_CTD`)

- **Typ**: IEC 61131-3 Funktionbaustein – Abwärtszähler (Counter Down)
- **Parameter**:
  - `PV = INT#10` ⟶ Preset-Wert = 10 (als Integer-Konstante)

- **Ereigniseingänge**:
  - `REQ` ⟶ wird bei steigender Flanke an `CD` oder `LD` angesteuert
- **Ereignisausgänge**:
  - `CNF` ⟶ wird ausgelöst, sobald der Baustein eine neue Berechnung abgeschlossen hat
- **Dateneingänge**:
  - `CD` (BOOL) ⟶ Zählimpuls (Count Down)
  - `LD` (BOOL) ⟶ Laden des Preset-Wertes
  - `PV` (INT) ⟶ Voreinstellwert (hier fest auf 10 gesetzt)
- **Datenausgänge**:
  - `Q` (BOOL) ⟶ wird `TRUE`, wenn der Zählerstand 0 erreicht hat
  - `CV` (INT) ⟶ aktueller Zählerstand

Funktionsweise: Der Baustein zählt bei jedem positiven Flankenwechsel an `CD` den internen Zähler um 1 herunter. Ein positiver Flanke an `LD` setzt den Zähler zurück auf den Wert von `PV`. Der Ausgang `Q` ist `TRUE`, solange der Zählerstand = 0 ist. Der aktuelle Zählerstand wird über `CV` ausgegeben.

`Input_CD` (Typ: `logiBUS::io::DI::logiBUS_IX`)

- **Typ**: Digitaleingang – physikalischer Eingang `Input_I1`
- **Parameter**:
  - `QI = TRUE` ⟶ Eingang aktiviert
  - `Input = Input_I1` ⟶ realer Eingang (z.B. Taster oder Sensor)
- **Ausgänge**:
  - `IND` (Ereignis) ⟶ wird bei Änderung des Eingangssignals ausgelöst
  - `IN` (BOOL) ⟶ aktueller Zustand des Eingangs

`Input_LD` (Typ: `logiBUS::io::DI::logiBUS_IX`)

- **Typ**: Digitaleingang – physikalischer Eingang `Input_I2`
- **Parameter**:
  - `QI = TRUE`
  - `Input = Input_I2`
- **Ausgänge**: wie `Input_CD`

`Output_Q1` (Typ: `logiBUS::io::DQ::logiBUS_QX`)

- **Typ**: Digitalausgang – physikalischer Ausgang `Output_Q1`
- **Parameter**:
  - `QI = TRUE`
  - `Output = Output_Q1` ⟶ realer Ausgang (z.B. Lampe oder Relais)
- **Eingänge**:
  - `REQ` (Ereignis) ⟶ wird bei neuem Zählergebnis ausgelöst
  - `OUT` (BOOL) ⟶ Wert für den Ausgang

`F_INT_TO_UDINT` (Typ: `iec61131::conversion::F_INT_TO_UDINT`)

- **Typ**: IEC-Konvertierungsfunktion von `INT` nach `UDINT`
- **Dateneingänge**:
  - `IN` (INT) ⟶ Eingangswert (hier der aktuelle Zählerstand)
- **Datenausgänge**:
  - `OUT` (UDINT) ⟶ konvertierter Wert (vorzeichenloser 32-Bit-Integer)

> **Hinweis**: Die Verwendung dieses Bausteins ist aus technischer Sicht nicht optimal, da der Zählerstand `CV` eines **Rückwärtszählers** nicht negativ werden kann (er stoppt bei 0). Eine direkte Kopplung ohne Typkonvertierung wäre möglich, der Baustein dient hier aber als didaktisches Beispiel für die Umwandlung von Datentypen.

`Q_NumericValue` (Typ: `isobus::UT::Q::Q_NumericValue`)

- **Typ**: Terminal-Ausgabebaustein zur Darstellung eines numerischen Wertes
- **Parameter**:
  - `u16ObjId = OutputNumber_N1` ⟶ Objekt-ID des numerischen Anzeigefeldes auf dem Terminal
- **Eingänge**:
  - `REQ` (Ereignis) ⟶ löst die Aktualisierung der Anzeige aus

  - `u32NewValue` (UDINT) ⟶ neuer darzustellender Wert
- **Funktionsweise**: Der Baustein sendet den übergebenen Wert an das konfigurierte Terminal-Feld, sodass der Benutzer den aktuellen Zählerstand auf einem Display ablesen kann.

Programmablauf und Verbindungen

Der Ablauf der Übung wird durch die Ereignis- und Datenverbindungen im FBNetzwerk bestimmt:

1. **Ereignisauslösung**
   - Die beiden Digitaleingänge `Input_CD` und `Input_LD` erzeugen bei einem Signalwechsel das Ereignis `IND`.
   - Beide Ereignisse werden auf den **gleichen** Ereigniseingang `REQ` des Zählers `FB_CTD` geschaltet. Das bedeutet: Jeder Tastendruck (egal ob CD oder LD) triggert eine Neuberechnung des Zählers.
2. **Datenkopplung**
   - Der **Zählimpuls** (`CD`) wird direkt vom Ausgang `IN` des Eingangsbausteins `Input_CD` an den Dateneingang `FB_CTD.CD` geführt.
   - Der **Ladeimpuls** (`LD`) wird vom Ausgang `IN` des `Input_LD` an `FB_CTD.LD` angeschlossen.
   - Der **Zählerausgang Q** wird an den Ausgabebaustein `Output_Q1.OUT` weitergegeben.
   - Der **aktuelle Zählerstand `CV`** wird über den Konvertierungsbaustein `F_INT_TO_UDINT` an das Terminal `Q_NumericValue.u32NewValue` gesendet.
3. **Terminal-Aktualisierung**
   - Sobald die Berechnung des Zählers abgeschlossen ist, wird das Ereignis `CNF` von `FB_CTD` ausgelöst.
   - Dieses Ereignis triggert sowohl den Ausgabebaustein `Output_Q1` als auch den Konvertierungsbaustein `F_INT_TO_UDINT`.
   - Nach der Konvertierung feuert `F_INT_TO_UDINT.CNF` und aktualisiert die numerische Anzeige auf dem Terminal.

Lernziele dieser Übung:

- Einsatz des IEC-Rückwärtszählers `FB_CTD` (Abwärtszähler) in 4diac.
- Kopplung von Hardwareeingängen und -ausgängen über logiBUS-Bausteine.
- Ausgabe eines Zählerwerts auf einem Terminal mithilfe eines isobus-Bausteins.
- Verständnis der Ereignis- und Datenflusssteuerung in einer SubApplikation.

Schwierigkeitsgrad: Einfach **Benötigte Vorkenntnisse**: Grundlegende Bedienung der 4diac-IDE, Verständnis von IEC 61131-3 Funktionbausteinen.

Starten der Übung: Laden Sie die Übung `Uebung_215` aus dem Paket `Uebungen` in ein leeres Projekt. Verbinden Sie die Hardwareeingänge `Input_I1` (CD) und `Input_I2` (LD) mit Tastern und den Ausgang `Output_Q1` mit einer Anzeige (z.B. Lampe). Das Terminal-Objekt `OutputNumber_N1` muss im Pool konfiguriert sein.

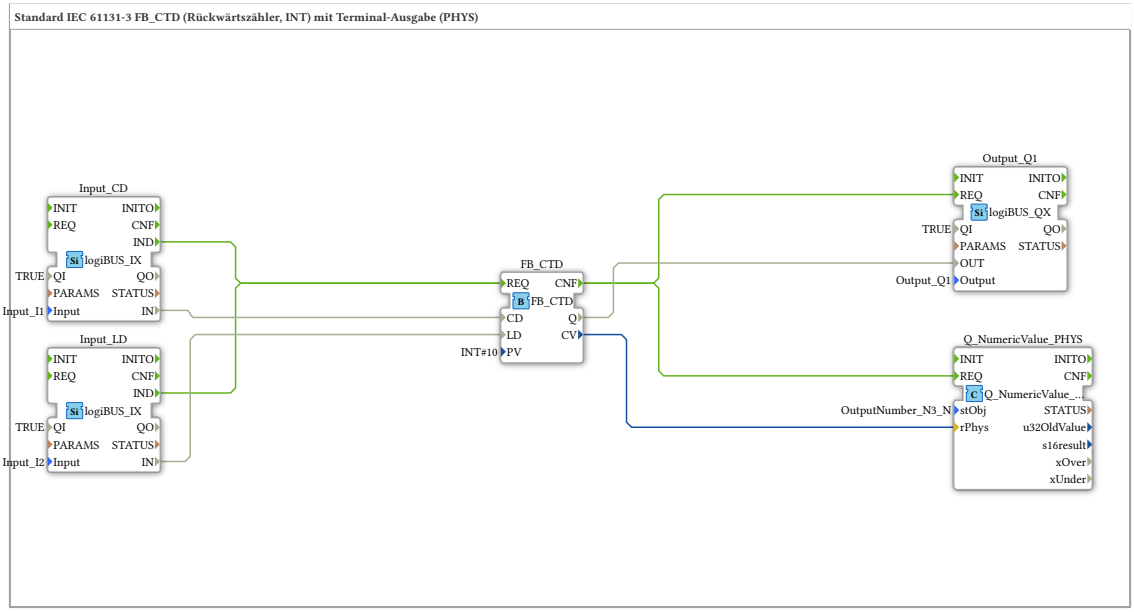
Zusammenfassung

In dieser Übung wurde ein vollständiger Rückwärtszähler gemäß IEC 61131-3 aufgebaut. Der Zähler zählt bei Tastendruck an `I1` von 10 abwärts, setzt sich bei Tastendruck an `I2` zurück und aktiviert den Ausgang `Q1`, sobald der Wert 0 erreicht ist. Der aktuelle Zählerstand wird auf einem Terminal ausgegeben. Die Übung demonstriert die Integration von IEC-Bausteinen mit logiBUS-Hardware und isobus-Visualisierung und vermittelt ein grundlegendes Verständnis für ereignisgesteuerte Automatisierungssysteme.

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

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

2.2.1.386 Uebung_215b: Standard IEC 61131-3 FB_CTD (Rückwärtszähler, INT) mit Terminal-Ausgabe (PHYS)



• – ◦ • – ◦ • – ◦ •

Einleitung

Diese Übung implementiert einen **Rückwärtszähler** gemäß IEC 61131-3 (Funktionsbaustein `FB_CTD`) mit einem Preset-Wert von **10**. Der Zähler wird über zwei digitale Eingänge (CD – Count Down, LD – Load) gesteuert. Der aktuelle Zählerstand wird auf einem Terminal (PHYS) ausgegeben, und der Ausgang Q zeigt an, ob der Zählerstand Null erreicht hat.

Verwendete Funktionsbausteine (FBs)

- **FB_CTD** (Typ: `iec61131::counters::FB_CTD`)
  - Parameter: `PV` = `INT#10` (Preset-Wert)
  - Ereignis-Eingang: `REQ` (Anforderung zur Ausführung)
  - Ereignis-Ausgang: `CNF` (Bestätigung der Verarbeitung)
  - Daten-Eingänge: `CD` (Count Down – Zählimpuls), `LD` (Load – Laden des Preset-Werts)
  - Daten-Ausgänge: `Q` (Ausgang, TRUE wenn CV = 0), `CV` (aktueller Zählerstand)
- **Input_CD** (Typ: `logiBUS::io::DI::logiBUS_IX`)
  - Parameter: `QI` = `TRUE`, `Input` = `Input_I1` (physischer Eingang I1)
  - Funktion: Digitaler Eingang, der bei einem Flankensignal ein Ereignis `IND` auslöst.
- **Input_LD** (Typ: `logiBUS::io::DI::logiBUS_IX`)
  - Parameter: `QI` = `TRUE`, `Input` = `Input_I2` (physischer Eingang I2)
  - Funktion: Digitaler Eingang für den Ladebefehl.
- **Output_Q1** (Typ: `logiBUS::io::DQ::logiBUS_QX`)
  - Parameter: `QI` = `TRUE`, `Output` = `Output_Q1` (physischer Ausgang Q1)
  - Funktion: Digitaler Ausgang, der den `Q`-Wert des Zählers auf eine physische Leitung ausgibt.
- **Q_NumericValue_PHYS** (Typ: `isobus::UT::Q::Q_NumericValue_PHYS`)
  - Parameter: `stObj` = `OutputNumber_N3` (Terminal-Ausgabestelle)
  - Funktion: Gibt den übergebenen numerischen Wert (`rPhys`) auf dem Terminal aus.

Programmablauf und Verbindungen

Die folgenden Ereignis- und Datenverbindungen definieren den Ablauf:

Ereignisverbindungen

- `Input_CD.IND` ⟶ `FB_CTD.REQ`: Bei einem Signal von Eingang I1 wird der Zählerbaustein getriggert (Count Down oder Load, abhängig vom LD-Signal).
- `Input_LD.IND` ⟶ `FB_CTD.REQ`: Ein Signal von Eingang I2 löst ebenfalls den Zählerbaustein aus (setzt CV auf PV).

- `FB_CTD.CNF` ⟶ `Output_Q1.REQ`: Nach der Verarbeitung wird der Ausgang Q aktualisiert.
- `FB_CTD.CNF` ⟶ `Q_NumericValue_PHYS.REQ`: Gleichzeitig wird der Zählerstand CV an das Terminal gesendet.

Datenverbindungen

- `Input_CD.IN` ⟶ `FB_CTD.CD`: Der Wert des Eingangs (TRUE/FALSE) wird als Count-Down-Signal übergeben.
- `Input_LD.IN` ⟶ `FB_CTD.LD`: Der Wert des Eingangs dient als Load-Signal.
- `FB_CTD.Q` ⟶ `Output_Q1.OUT`: Der Ausgangswert Q (TRUE bei CV=0) wird auf den physischen Ausgang Q1 gelegt.
- `FB_CTD.CV` ⟶ `Q_NumericValue_PHYS.rPhys`: Der aktuelle Zählerstand (INT) wird direkt an den Terminalbaustein übergeben. Ein Kommentar im Netzwerk weist darauf hin, dass **INT ohne Konvertierung auf REAL** geschlossen werden kann.

Ablaufbeschreibung

- **Load**: Wenn Eingang I2 aktiv wird, setzt der Zähler seinen internen Wert `CV` auf den Preset-Wert (10). Der Ausgang `Q` wird zurückgesetzt (FALSE).
- **Count Down**: Bei jedem positiven Flanke an Eingang I1 wird `CV` um 1 dekrementiert. Sobald `CV = 0` erreicht, wird `Q` auf TRUE gesetzt und bleibt TRUE bis zum nächsten Load.
- Die Ausgabe `Q` steuert den physischen Ausgang Q1, der dann z. B. eine Lampe oder ein Signal schalten kann.
- Der aktuelle Zählerstand wird stets nach jeder Verarbeitung auf dem Terminal (PHYS) angezeigt.

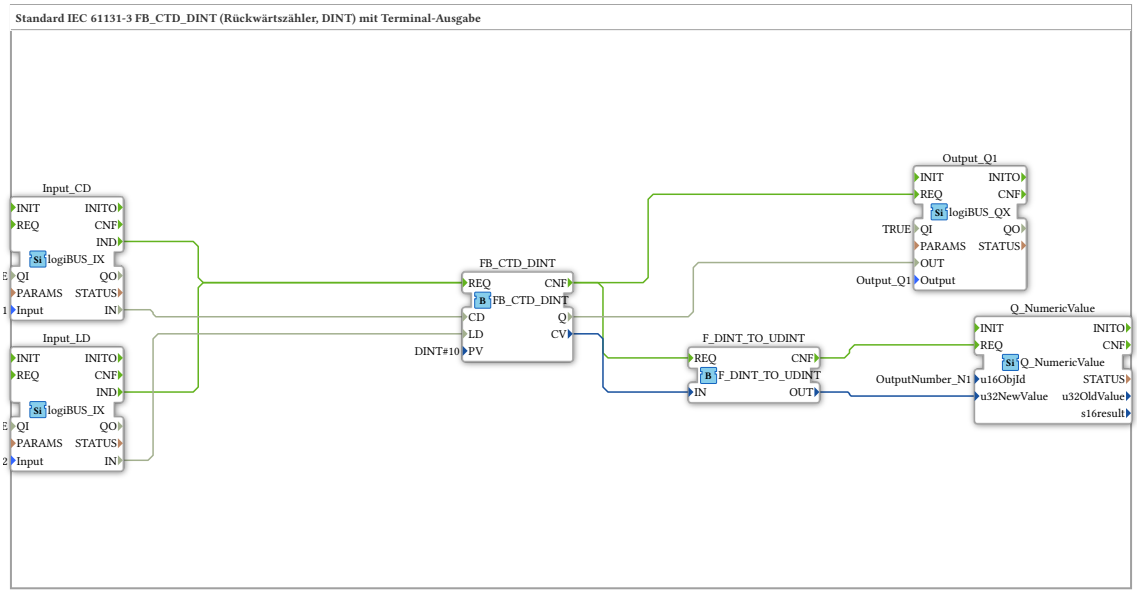
Zusammenfassung

Die Übung 215b demonstriert den Einsatz des IEC 61131-3 Rückwärtszählers `FB_CTD` in einer 4diac-IDE-Umgebung. Durch die Kombination von digitalen Eingängen (I1, I2), einem digitalen Ausgang (Q1) und einer Terminalausgabe lernen Sie die typische Nutzung eines Zählers in der Automatisierungstechnik kennen. Die direkte Verbindung des Integer-Zählerstands an eine Terminalausgabestelle zeigt die flexible Datenkonvertierung des Systems.

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

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

2.2.1.387 Uebung_216: Standard IEC 61131-3 FB_CTD_DINT (Rückwärtszähler, DINT) mit Terminal-Ausgabe



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert die Implementierung eines **Rückwärtszählers (CTD)** nach IEC 61131-3 mit ganzzahligem Datentyp `DINT` (Double Integer) und einer **Terminal-Ausgabe** des aktuellen Zählerstandes. Der Zähler wird durch zwei digitale Eingänge gesteuert:

- **Eingang I1** – dekrementiert den Zähler bei positiver Flanke.
- **Eingang I2** – lädt den vorgegebenen Startwert (PV = 10) in den Zähler.

Ein digitaler Ausgang **Q1** wird aktiviert, sobald der Zählerstand den Wert 0 erreicht hat. Gleichzeitig wird der aktuelle Zählerstand über einen Terminal-Baustein ausgegeben.

Verwendete Funktionsbausteine (FBs)

- **FB_CTD_DINT** – `iec61131::counters::FB_CTD_DINT`
  - Parameter: `PV` = `DINT#10` (Startwert)
  - Ereigniseingang: `REQ` (Zählimpuls von CD oder LD)
  - Ereignisausgang: `CNF` (Bestätigung nach Zählvorgang)
  - Dateneingänge: `CD` (Dekrement-Eingang), `LD` (Lade-Eingang)
  - Datenausgänge: `Q` (Signal bei Erreichen von 0), `CV` (aktueller Zählerstand)
- **Input_CD** – `logiBUS::io::DI::logiBUS_IX`
  - Parameter: `QI` = `TRUE`, `Input` = `Input_I1`
  - Liefert den Zustand des digitalen Eingangs I1.
- **Input_LD** – `logiBUS::io::DI::logiBUS_IX`
  - Parameter: `QI` = `TRUE`, `Input` = `Input_I2`
  - Liefert den Zustand des digitalen Eingangs I2.
- **Output_Q1** – `logiBUS::io::DQ::logiBUS_QX`
  - Parameter: `QI` = `TRUE`, `Output` = `Output_Q1`
  - Setzt den digitalen Ausgang Q1, wenn der Zähler Q-Ausgang aktiv ist.
- **F_DINT_TO_UDINT** – `iec61131::conversion::F_DINT_TO_UDINT`
  - Wandelt den Zählerstand von `DINT` (vorzeichenbehaftet) in `UDINT` (vorzeichenlos) um.
  - **Hinweis:** Negative Zählerstände sind nach dieser Konvertierung nicht mehr darstellbar – die Übung macht dies als didaktische Einschränkung bewusst.
- **Q_NumericValue** – `isobus::UT::Q::Q_NumericValue`
  - Parameter: `u16ObjId` = `OutputNumber_N1`
  - Gibt einen numerischen Wert auf dem Terminal aus.

Programmablauf und Verbindungen

1. **Ereignisverkettung**
   - Eingang I1 oder I2 löst über den `IND`-Ereignisausgang den `REQ`-Eingang des Zählers aus.
   - Nach erfolgreicher Verarbeitung (CNF) werden gleichzeitig der Ausgangsbaustein `Output_Q1` und die Konvertierung `F_DINT_TO_UDINT` angestoßen.
   - Nach der Konvertierung wird der Wert an den Terminal-Baustein `Q_NumericValue` übergeben.
2. **Datenverkettung**
   - `Input_CD.IN` ⟶ `FB_CTD_DINT.CD` (Dekrement)
   - `Input_LD.IN` ⟶ `FB_CTD_DINT.LD` (Laden)
   - `FB_CTD_DINT.Q` ⟶ `Output_Q1.OUT` (Setzen des Ausgangs bei Zählerstand 0)
   - `FB_CTD_DINT.CV` ⟶ `F_DINT_TO_UDINT.IN` (Aktueller Zählerstand)
   - `F_DINT_TO_UDINT.OUT` ⟶ `Q_NumericValue.u32NewValue` (Terminalausgabe)
3. **Funktionsweise**
   - Bei jeder positiven Flanke an I1 wird der Zählerstand um 1 verringert.
   - Bei einer positiven Flanke an I2 wird der Zähler mit dem Wert 10 (PV) geladen.
   - Sobald der Zählerstand 0 erreicht, wird der Ausgang Q1 gesetzt.
   - Der aktuelle Zählerstand wird fortlaufend auf dem Terminal ausgegeben.

Didaktischer Hinweis: Die Konvertierung `DINT_TO_UDINT` ist für negative Zählerstände nicht geeignet (UDINT kann nur positive Werte darstellen). Dies ist als **bewusste Einschränkung** in die Übung eingebaut, um auf die Problematik der Datentypumwandlung hinzuweisen.

Zusammenfassung

Die Übung „Uebung_216“ vermittelt den Einsatz eines IEC 61131-3 Rückwärtszählers (`FB_CTD_DINT`) in Verbindung mit einer Terminalausgabe. Sie zeigt:

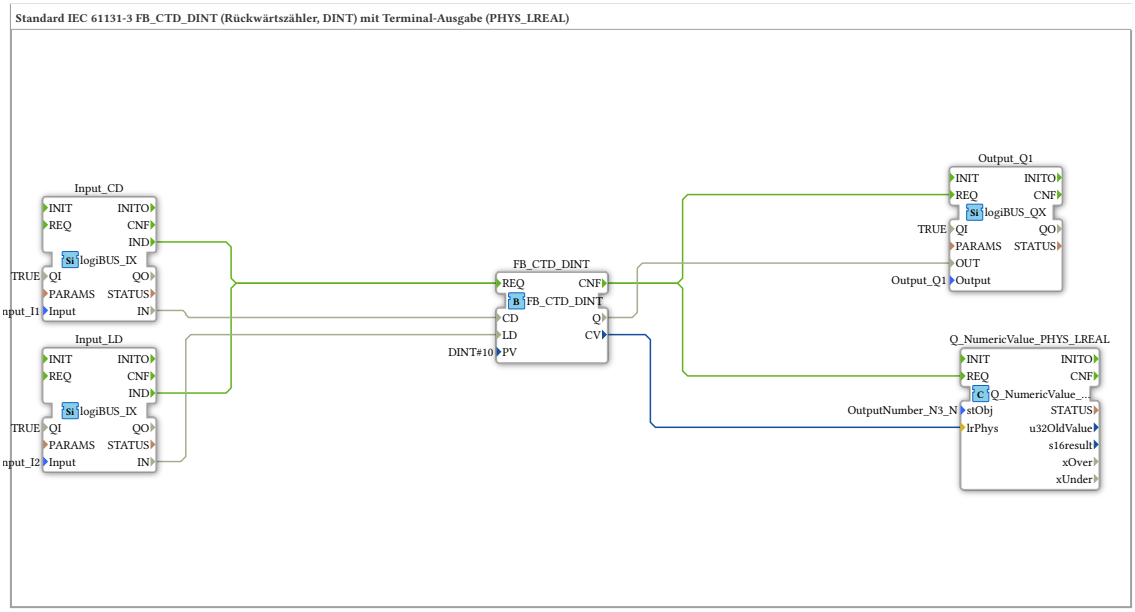
- Die Steuerung eines Zählers über zwei digitale Eingänge (Dekrement/Laden).
- Die Verwendung eines Ausgangsbausteins zur Signalisierung des Erreichens des Zählendes.
- Die Umwandlung von Datentypen (`DINT` → `UDINT`) und deren Grenzen (keine negativen Werte).
- Die Visualisierung von Zählerwerten auf einem Terminal.

Die Übung ist für Einsteiger in die 4diac-IDE geeignet, die bereits grundlegende Kenntnisse über IEC 61131-3-Bausteine und digitale Ein-/Ausgänge besitzen. Sie kann direkt geladen und mit simulierten oder realen Eingängen getestet werden.

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

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

2.2.1.388 Uebung_216b: Standard IEC 61131-3 FB_CTD_DINT (Rückwärtszähler, DINT) mit Terminal-Ausgabe (PHYS_LREAL)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung implementiert einen **Rückwärtszähler (CTD)** gemäß IEC 61131-3 mit dem Datentyp `DINT` und einer Terminalausgabe des aktuellen Zählwerts als physikalische Größe (`PHYS_LREAL`). Der Zähler wird über zwei digitale Eingänge gesteuert (**CD** = Count Down, **LD** = Load) und gibt das Zählersignal (`Q`) auf einen digitalen Ausgang aus. Parallel dazu wird der aktuelle Zählerstand über einen numerischen Ausgangsbaustein auf einem Terminal visualisiert.

Verwendete Funktionsbausteine (FBs)

- **FB_CTD_DINT** – Typ: `iec61131::counters::FB_CTD_DINT`
  - Parameter: `PV` = `DINT#10` (Vorwahlwert 10)
  - Zählt bei jedem steigenden Flanke an `CD` von `PV` abwärts; wird bei Aktivierung von `LD` auf den Wert `PV` zurückgesetzt.
- **Input_CD** – Typ: `logiBUS::io::DI::logiBUS_IX`
  - Parameter: `QI` = `TRUE`, `Input` = `Input_I1` (Hardware-Eingang I1)

  - Stellt den Zähleingang (Count Down) bereit.
- **Input_LD** – Typ: `logiBUS::io::DI::logiBUS_IX`
  - Parameter: `QI` = `TRUE`, `Input` = `Input_I2` (Hardware-Eingang I2)
  - Stellt den Ladeeingang (Load) bereit.
- **Output_Q1** – Typ: `logiBUS::io::DQ::logiBUS_QX`
  - Parameter: `QI` = `TRUE`, `Output` = `Output_Q1` (Hardware-Ausgang Q1)
  - Gibt den logischen Zustand des Zählerausgangs `Q` aus.
- **Q_NumericValue_PHYS_LREAL** – Typ: `isobus::UT::Q::Q_NumericValue_PHYS_LREAL`
  - Parameter: `stObj` = `OutputNumber_N3` (Terminal-Objekt für die Anzeige)
  - Wandelt den aktuellen Zählwert (`CV`) in eine physikalische `LREAL`-Größe um und zeigt ihn auf dem Terminal an.

Programmablauf und Verbindungen

Die Steuerung erfolgt rein ereignisgesteuert über die **IND**-Ereignisse der Eingänge:

1. **Zähleingang (CD):** Eine steigende Flanke an `Input_I1` wird über den Baustein `Input_CD` erfasst und löst das Ereignis `IND` aus. Dieses wird mit dem `REQ`-Ereignis des Zählers `FB_CTD_DINT` verbunden. Gleichzeitig wird der physikalische Eingangswert (`IN`) über die Datenverbindung `Input_CD.IN` → `FB_CTD_DINT.CD` an den Zähleingang übertragen.
2. **Ladeeingang (LD):** Analog dazu wird eine steigende Flanke an `Input_I2` über `Input_LD` erfasst und ebenfalls mit dem `REQ` des Zählers verbunden. Der Eingangswert (`IN`) wird an den Ladeeingang `FB_CTD_DINT.LD` weitergeleitet. *Hinweis*: Beide Ereignisse (CD und LD) nutzen dasselbe `REQ`-Ereignis des Zählers. Der Zähler wertet intern die jeweiligen Datenleitungen aus, um die Operation (zählen oder laden) zu unterscheiden.
3. **Ausgang Q1 und Terminalanzeige:** Nach Abschluss der Zählerverarbeitung wird das `CNF`-Ereignis ausgelöst. Dieses wird parallel an die `REQ`-Eingänge von `Output_Q1` und `Q_NumericValue_PHYS_LREAL` gesendet.
   - Der Ausgangswert `FB_CTD_DINT.Q` (logisch, wenn Zählerstand ≤ 0) wird über die Datenverbindung an `Output_Q1.OUT` gelegt und somit am Hardware-Ausgang Q1 ausgegeben.
   - Der aktuelle Zählerstand `FB_CTD_DINT.CV` (Typ `DINT`) wird an `Q_NumericValue_PHYS_LREAL.lrPhys` übergeben und als physikalischer `LREAL`-Wert auf dem Terminal dargestellt.

Zusammenfassung der Verbindungen:

Sender	Empfänger	Art
`Input_CD.IND`	`FB_CTD_DINT.REQ`	Ereignis
`Input_LD.IND`	`FB_CTD_DINT.REQ`	Ereignis
`FB_CTD_DINT.CNF`	`Output_Q1.REQ`	Ereignis
`FB_CTD_DINT.CNF`	`Q_NumericValue_PHYS_LREAL.REQ`	Ereignis
`Input_CD.IN`	`FB_CTD_DINT.CD`	Daten
`Input_LD.IN`	`FB_CTD_DINT.LD`	Daten
`FB_CTD_DINT.Q`	`Output_Q1.OUT`	Daten
`FB_CTD_DINT.CV`	`Q_NumericValue_PHYS_LREAL.lrPhys`	Daten

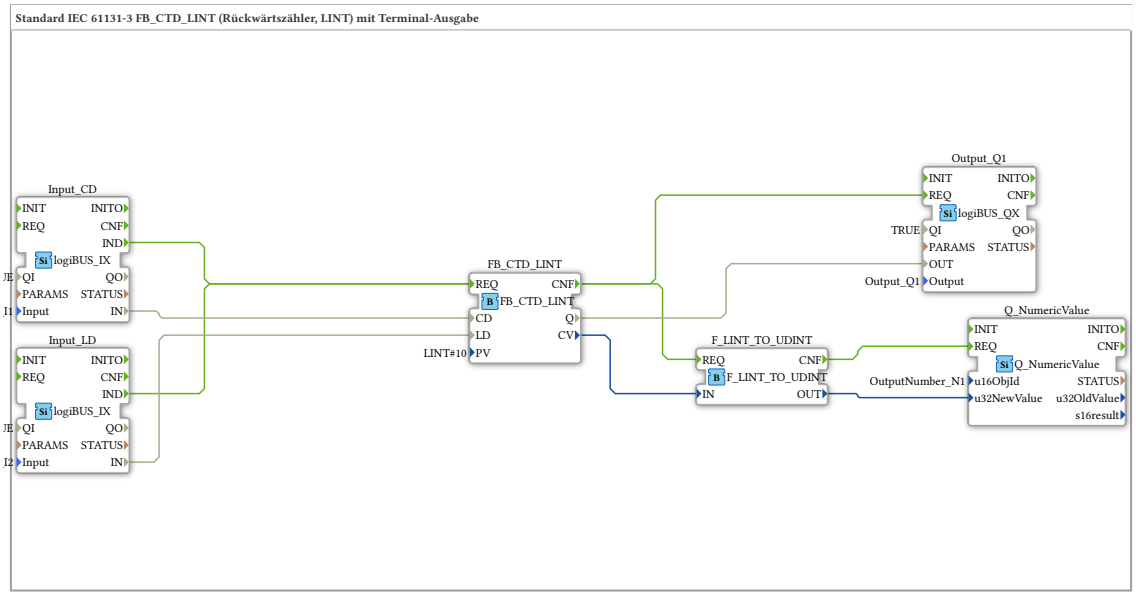
Zusammenfassung

Die Übung 216b demonstriert die Anwendung eines IEC 61131-3 Rückwärtszählers (`FB_CTD_DINT`) in der 4diac-IDE. Der Zähler wird über zwei digitale Eingänge gesteuert und gibt sowohl ein binäres Signal (Zähler erreicht Null) als auch eine numerische Anzeige des aktuellen Zählwerts auf einem Terminal aus. Die Ereignissteuerung stellt sicher, dass die Ausgangswerte nur nach einer vollständigen Zähleroperation aktualisiert werden. Dieses Beispiel zeigt grundlegende Konzepte der ereignisgesteuerten Applikationsentwicklung mit Standard-Funktionsbausteinen und Hardware-Schnittstellen.

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

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

2.2.1.389 Uebung_217: Standard IEC 61131-3 FB_CTD_LINT (Rückwärtszähler, LINT) mit Terminal-Ausgabe



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung implementiert einen Rückwärtszähler (Count Down, CTD) nach IEC 61131-3 mit dem Datentyp LINT (64-Bit Integer). Der Zähler wird durch zwei digitale Eingänge gesteuert: ein Count-Down-Impuls (CD) und ein Load-Impuls (LD). Der aktuelle Zählerstand (CV) wird über eine Konvertierung in den Typ UDINT (Unsigned 32-Bit) an eine numerische Terminalausgabe gesendet. Der Ausgang Q signalisiert, wenn der Zählerstand ≤ 0 ist.

Ein Kommentar im Netzwerk weist darauf hin, dass die Konvertierung `F_LINT_TO_UDINT` für negative Zählerstände ungeeignet ist, da UDINT keine negativen Zahlen darstellen kann.

Verwendete Funktionsbausteine (FBs)

FB_CTD_LINT (Rückwärtszähler LINT)

- **Typ**: `iec61131::counters::FB_CTD_LINT`
- **Parameter**: `PV = LINT#10` (Presetwert, Anfangswert für den Zähler)
- **Ereigniseingänge**: `REQ` (wird sowohl von `Input_CD` als auch von `Input_LD` getriggert)
- **Ereignisausgänge**: `CNF` (Bestätigung nach Verarbeitung)
- **Dateneingänge**: `CD` (Count-Down-Signal), `LD` (Load-Signal)
- **Datenausgänge**: `CV` (aktueller Zählerstand, LINT), `Q` (Ausgang, wenn CV ≤ 0)

Input_CD (Digitaler Eingang für Count-Down)

- **Typ**: `logiBUS::io::DI::logiBUS_IX`
- **Parameter**: `QI = TRUE`, `Input = Input_I1` (Hardware-Adresse)
- **Ereignisausgänge**: `IND` (Ereignis bei Signalwechsel)
- **Datenausgänge**: `IN` (aktueller Eingangswert, BOOL)

Input_LD (Digitaler Eingang für Load)

- **Typ**: `logiBUS::io::DI::logiBUS_IX`
- **Parameter**: `QI = TRUE`, `Input = Input_I2`
- **Ereignisausgänge**: `IND`
- **Datenausgänge**: `IN`

Output_Q1 (Digitaler Ausgang)

- **Typ**: `logiBUS::io::DQ::logiBUS_QX`
- **Parameter**: `QI = TRUE`, `Output = Output_Q1`
- **Ereigniseingänge**: `REQ`
- **Dateneingänge**: `OUT` (setzt den Ausgang)

F_LINT_TO_UDINT (Konvertierung LINT → UDINT)

- **Typ**: `iec61131::conversion::F_LINT_TO_UDINT`
- **Ereigniseingänge**: `REQ`
- **Ereignisausgänge**: `CNF`
- **Dateneingänge**: `IN` (LINT)
- **Datenausgänge**: `OUT` (UDINT) – Achtung: Negative Eingangswerte werden nicht korrekt abgebildet.

Q_NumericValue (Terminal-Ausgabe numerischer Wert)

- **Typ**: `isobus::UT::Q::Q_NumericValue`
- **Parameter**: `u16ObjId = OutputNumber_N1` (Objektkennung im Terminal)
- **Ereigniseingänge**: `REQ`
- **Dateneingänge**: `u32NewValue` (UDINT, neuer anzuzeigender Wert)

Programmablauf und Verbindungen

1. **Ereignisverkettung**:
   - Ein Eingangsereignis von `Input_CD.IND` oder `Input_LD.IND` triggert den `REQ` des Zählers `FB_CTD_LINT`.
   - Nach der Verarbeitung des Zählers (Ausgang `CNF`) werden gleichzeitig der Ausgang `Output_Q1` (über `REQ`) und die Konvertierung `F_LINT_TO_UDINT` (über `REQ`) aufgerufen.
   - Nach Abschluss der Konvertierung (`CNF`) wird die Terminalausgabe `Q_NumericValue` (über `REQ`) aktualisiert.
2. **Datenverbindungen**:
   - `Input_CD.IN` → `FB_CTD_LINT.CD`: Der Wert des digitalen Eingangs I1 steuert, ob der Zähler herunterzählt.
   - `Input_LD.IN` → `FB_CTD_LINT.LD`: Der Wert des digitalen Eingangs I2 lädt den Presetwert (PV) in den Zähler.
   - `FB_CTD_LINT.Q` → `Output_Q1.OUT`: Das Ausgangssignal des Zählers wird direkt auf den digitalen Ausgang Q1 gelegt.
   - `FB_CTD_LINT.CV` → `F_LINT_TO_UDINT.IN`: Der aktuelle Zählerstand (LINT) wird zur Konvertierung weitergeleitet.
   - `F_LINT_TO_UDINT.OUT` → `Q_NumericValue.u32NewValue`: Der konvertierte Wert (UDINT) wird zur Anzeige im Terminal gesendet.
3. **Hinweis zur Konvertierung**: Die Verwendung von `F_LINT_TO_UDINT` ist für negative Zählerstände nicht geeignet, da der Wertebereich von UDINT nur nicht-negative Zahlen umfasst. Bei negativem Zählerstand wird ein unerwartetes Ergebnis angezeigt oder die Konvertierung kann fehlschlagen. In der Praxis sollte eine andere Darstellung (z. B. Vorzeichenbehaftet) verwendet werden.

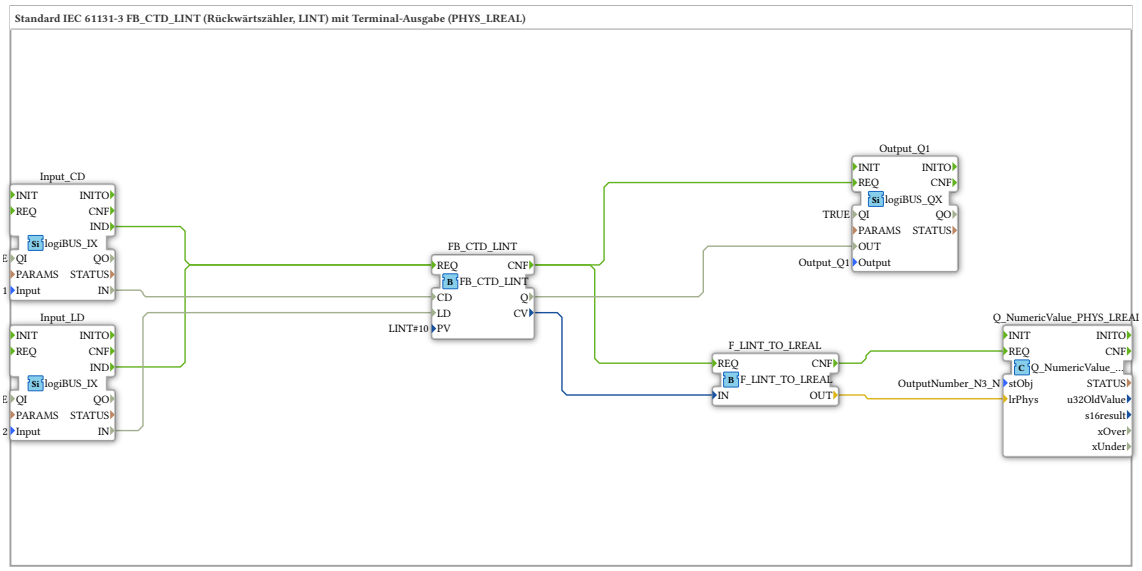
Zusammenfassung

In dieser Übung wird ein IEC 61131-3 Rückwärtszähler (`FB_CTD_LINT`) mit zwei digitalen Eingängen angesteuert. Der Zählerstand wird auf einem Terminal ausgegeben, wobei die Konvertierung von LINT nach UDINT bewusst eine Einschränkung für negative Werte darstellt. Die Aufgabe veranschaulicht die ereignisgesteuerte Verarbeitung in 4diac, die Kopplung von Hardwareeingängen und die Grenzen von Datentypkonvertierungen.

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

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

2.2.1.390 Uebung_217b: Standard IEC 61131-3 FB_CTD_LINT (Rückwärtszähler, LINT) mit Terminal-Ausgabe (PHYS_LREAL)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung implementiert einen Rückwärtszähler nach IEC 61131-3 (Typ `FB_CTD_LINT`), der mit einem LINT-Datentyp arbeitet. Der aktuelle Zählerstand wird über einen physikalischen LREAL-Ausgang auf einem Terminal dargestellt. Die Übung demonstriert die Verwendung eines IEC-Standardzählers, die Ankopplung an reale Ein-/Ausgänge (logiBUS) und die Datentypkonvertierung von LINT nach LREAL für die Terminalausgabe.

Verwendete Funktionsbausteine (FBs)

Folgende Funktionsbausteine werden im Netzwerk der SubApp verwendet:

- **FB_CTD_LINT** (Typ: `iec61131::counters::FB_CTD_LINT`)
  - Parameter: `PV` = `LINT#10` (Presetwert = 10)
  - Eingänge: Ereignis `REQ`, Daten `CD` (Zählimpuls), `LD` (Laden des Presetwerts)
  - Ausgänge: Ereignis `CNF`, Daten `Q` (Zählerstand > 0), `CV` (aktueller Zählerstand)
- **Input_CD** (Typ: `logiBUS::io::DI::logiBUS_IX`)
  - Parameter: `QI` = `TRUE`, `Input` = `Input_I1` (physischer Digitaleingang 1)
  - Ausgang: Daten `IN` (Bool)
- **Input_LD** (Typ: `logiBUS::io::DI::logiBUS_IX`)
  - Parameter: `QI` = `TRUE`, `Input` = `Input_I2` (physischer Digitaleingang 2)
  - Ausgang: Daten `IN` (Bool)
- **Output_Q1** (Typ: `logiBUS::io::DQ::logiBUS_QX`)
  - Parameter: `QI` = `TRUE`, `Output` = `Output_Q1` (physischer Digitalausgang 1)
  - Eingang: Daten `OUT` (Bool)
- **F_LINT_TO_LREAL** (Typ: `iec61131::conversion::F_LINT_TO_LREAL`)
  - Eingang: Daten `IN` (LINT)
  - Ausgang: Daten `OUT` (LREAL)
- **Q_NumericValue_PHYS_LREAL** (Typ: `isobus::UT::Q::Q_NumericValue_PHYS_LREAL`)
  - Parameter: `stObj` = `OutputNumber_N3` (Terminalausgabe-Objekt)
  - Eingang: Daten `lrPhys` (LREAL)

Programmablauf und Verbindungen

Die Übung arbeitet ereignisgesteuert:

1. **Ereignispfad**:
   - Ein steigender Impuls am Digitaleingang `Input_I1` (verbunden mit `Input_CD`) erzeugt ein Ereignis `IND`.
   - Ebenso erzeugt ein Impuls an `Input_I2` (verbunden mit `Input_LD`) ein `IND`-Ereignis.
   - Beide Ereignisse werden auf den Ereigniseingang `REQ` des Zählers `FB_CTD_LINT` geführt.
   - Nach Verarbeitung des Zählers (Ausgang `CNF`) wird einerseits der Ausgang `Output_Q1` (über `REQ`) und andererseits die Konvertierung `F_LINT_TO_LREAL` (über `REQ`) getriggert.
   - Nach der Konvertierung wird das Ereignis an die Terminalausgabe `Q_NumericValue_PHYS_LREAL` weitergeleitet.
2. **Datenpfad**:
   - Der digitale Wert von `Input_CD.IN` (Bool) wird auf den Dateneingang `CD` des Zählers gelegt.
   - Der digitale Wert von `Input_LD.IN` wird auf den Dateneingang `LD` des Zählers gelegt.
   - Der Zählerausgang `Q` (Bool) wird auf den Dateneingang `OUT` des Ausgangsbausteins `Output_Q1` geführt.
   - Der aktuelle Zählerstand `CV` (LINT) wird an den Konverter `F_LINT_TO_LREAL.IN` übergeben.
   - Der konvertierte Wert (LREAL) wird an den Terminalbaustein `Q_NumericValue_PHYS_LREAL.lrPhys` gesendet.

Funktionsweise des Zählers:

- Solange kein Ladesignal (`LD` = FALSE) anliegt, zählt der Baustein bei jedem steigenden Impuls an `CD` von 10 rückwärts (Presetwert = `PV` = 10).
- Ein Ladesignal setzt den aktuellen Zählerstand auf den Wert von `PV` zurück.
- Der Ausgang `Q` ist `TRUE`, solange der Zählerstand größer als 0 ist; bei Erreichen von 0 wird `Q` = `FALSE` (Überlauf ist nicht definiert, bleibt bei 0).
- Der aktuelle Zählerstand wird auf dem Terminal als physikalischer LREAL-Wert ausgegeben.

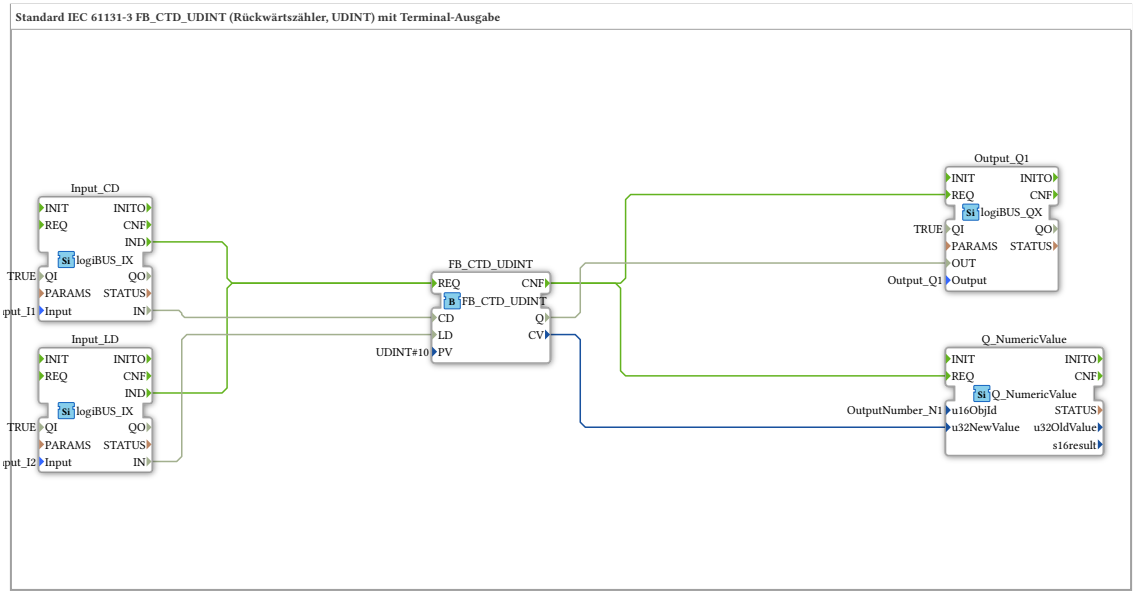
Zusammenfassung

Die Übung **Uebung_217b** realisiert einen standardkonformen Rückwärtszähler (`FB_CTD_LINT`) mit Terminalausgabe. Sie verknüpft digitale Eingänge (logiBUS) als Zähl- und Ladeimpulse, einen Digitalausgang als Meldeausgang und eine LINT-zu-LREAL-Konvertierung für die Anzeige des aktuellen Zählerstands auf einem Terminal. Der Ablauf ist vollständig ereignisgesteuert und zeigt die Integration von IEC 61131-3-Bausteinen mit logiBUS-I/O und Terminalausgaben in der 4diac-IDE.

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

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

2.2.1.391 Uebung_218: Standard IEC 61131-3 FB_CTD_UDINT (Rückwärtszähler, UDINT) mit Terminal-Ausgabe



• – ○ • – ○ • – ○ •

Einleitung

In dieser Übung wird ein Rückwärtszähler nach IEC 61131-3 (Typ `FB_CTD_UDINT`) implementiert. Der Funktionsbaustein zählt bei jedem negativen Flanke am Eingang `CD` (Count Down) rückwärts, beginnend vom voreingestellten Wert `PV` (Preset Value). Über einen Ladeeingang (`LD`) kann der Zähler auf den Startwert zurückgesetzt werden. Der aktuelle Zählerstand (`CV`) wird auf ein Terminal ausgegeben, und der Ausgang `Q` wird gesetzt, sobald der Zählerstand Null erreicht.

Die Hardware-Anbindung erfolgt über zwei digitale Eingänge (I1, I2) und einen digitalen Ausgang (Q1) sowie eine Ausgabe für numerische Werte auf dem Terminal. Die gesamte Konfiguration ist als SubApplikation in 4diac IDE umgesetzt.

Verwendete Funktionsbausteine (FBs)

FB-Name	Typ	Beschreibung
`FB_CTD_UDINT`	`iec61131::counters::FB_CTD_UDINT`	IEC 61131-3 Rückwärtszähler mit Eingängen `CD`, `LD`, `PV` und Ausgängen `Q` und `CV`. Parameter: `PV = UDINT#10` (Startwert 10).
`Input_CD`	`logiBUS::io::DI::logiBUS_IX`	Digitaler Eingang für das Zählsignal (Taster I1). Parameter: `QI=TRUE`, `Input=Input_I1`.
`Input_LD`	`logiBUS::io::DI::logiBUS_IX`	Digitaler Eingang für das Laden des Zählers (Taster I2). Parameter: `QI=TRUE`, `Input=Input_I2`.
`Output_Q1`	`logiBUS::io::DQ::logiBUS_QX`	Digitaler Ausgang für die Anzeige „Zählerstand = 0” (Lampe Q1). Parameter: `QI=TRUE`, `Output=Output_Q1`.
`Q_NumericValue`	`isobus::UT::Q::Q_NumericValue`	Terminal-Ausgabe für den aktuellen Zählerstand (`CV`). Parameter: `u16ObjId=OutputNumber_N1`.

Hinweis: Im Netzwerk befindet sich ein Kommentar, der darauf hinweist, dass eine explizite Typkonvertierung (`F_UDINT_TO_UDINT`) nicht notwendig ist, da der Datenfluss direkt erfolgt.

Programmablauf und Verbindungen

Ereignissteuerung

1. **Ereignisquelle:**
   - `Input_CD.IND` (Signal von Taster I1) wird mit `FB_CTD_UDINT.REQ` verbunden.
   - `Input_LD.IND` (Signal von Taster I2) wird ebenfalls mit `FB_CTD_UDINT.REQ` verbunden.
2. **Ereignissenke:**
   - `FB_CTD_UDINT.CNF` (Bestätigung des Zählers) löst zwei Aktionen aus:
     - `Output_Q1.REQ` (Aktualisierung des digitalen Ausgangs Q1).
     - `Q_NumericValue.REQ` (Aktualisierung der Terminal-Anzeige).

Datenflüsse

Quelle	Ziel	Bedeutung
`Input_CD.IN`	`FB_CTD_UDINT.CD`	Taster I1 als Zählimpuls (Rückwärtszählen).
`Input_LD.IN`	`FB_CTD_UDINT.LD`	Taster I2 als Ladesignal (Setzen auf PV).
`FB_CTD_UDINT.Q`	`Output_Q1.OUT`	Ausgang Q1 wird aktiv, sobald Zählerstand = 0 ist.
`FB_CTD_UDINT.CV`	`Q_NumericValue.u32NewValue`	Aktueller Zählerstand (als 32-Bit-Wert) an das Terminal.

Funktionsweise

- Nach dem Start wird der Zähler durch das Ladesignal (`LD`) auf den voreingestellten Wert `PV = 10` gesetzt.
- Jedes negativ werdende Signal am `CD`-Eingang (Taster I1) verringert den Zähler um 1.
- Sobald der Zählerstand null erreicht, wird der Ausgang `Q` auf `TRUE` gesetzt und der digitale Ausgang Q1 (Lampe) leuchtet.
- Der aktuelle Zählerstand (`CV`) wird fortlaufend auf dem Terminal (Objekt `OutputNumber_N1`) ausgegeben.

Lernziele & Vorkenntnisse

- **Schwierigkeitsgrad:** Einfach
- **Vorkenntnisse:** Grundkenntnisse in IEC 61131-3 und Umgang mit der 4diac IDE.
- **Lernziele:**
  - Verwendung des Standard-Rückwärtszählers `FB_CTD_UDINT`.
  - Anbindung digitaler Ein-/Ausgänge und einer Terminal-Ausgabe.
  - Verständnis von Ereignis- und Datenverbindungen in 4diac.

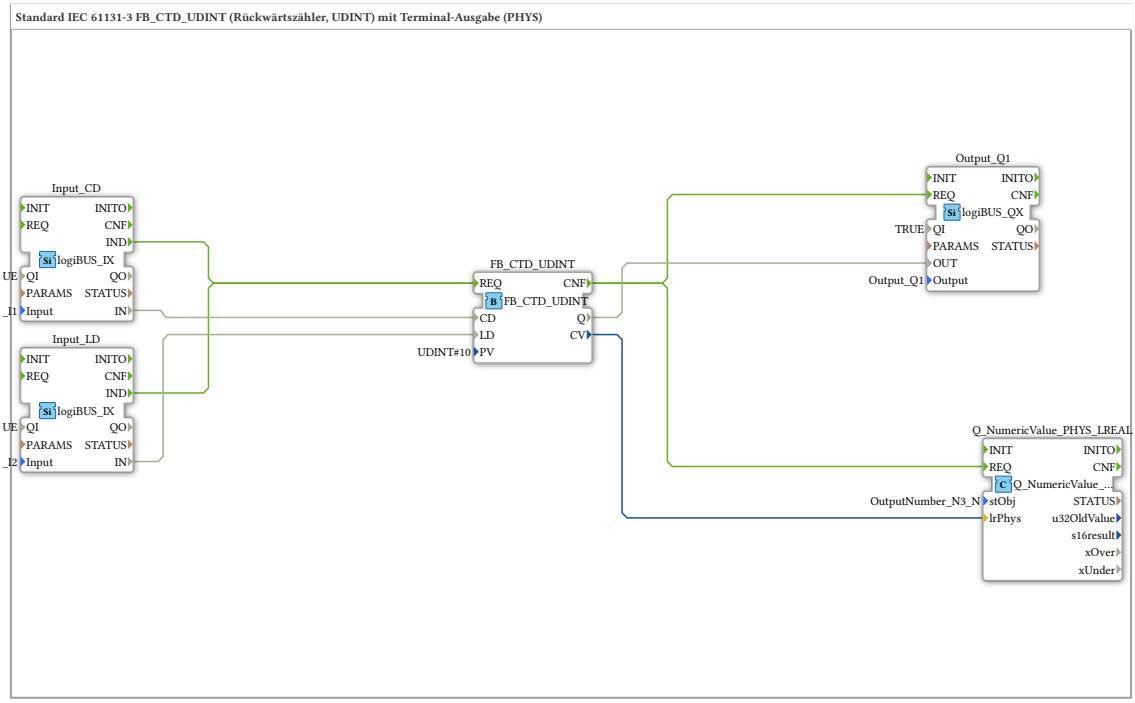
Zusammenfassung

Die Übung 218 realisiert einen vollständigen Rückwärtszähler mit IEC 61131-3 Standardbaustein. Durch die Kombination von zwei Tastern (Zählen und Laden) sowie einer Ausgabe auf das Terminal und eine Lampe wird das Verhalten des Zählers anschaulich demonstriert. Die Integration in 4diac erfolgt durch einfache Ereignis- und Datenverknüpfungen, die eine robuste und erweiterbare Steuerung ermöglichen.

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

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

2.2.1.392 Uebung_218b: Standard IEC 61131-3 FB_CTD_UDINT (Rückwärtszähler, UDINT) mit Terminal-Ausgabe (PHYS)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung implementiert einen Rückwärtszähler (Down Counter) nach IEC 61131-3 vom Typ **FB_CTD_UDINT** (Zählwert als UDINT). Der Zähler wird über zwei digitale Eingänge gesteuert: Ein Impuls an **Eingang I1** (CD) zählt den aktuellen Wert um 1 herunter, ein Impuls an **Eingang I2** (LD) lädt den voreingestellten Wert (PV) in den Zähler. Der aktuelle Zählerstand wird sowohl als **physikalischer Wert (LREAL)** über ein Terminal ausgegeben (FB `Q_NumericValue_PHYS_LREAL`) als auch als **digitales Signal (Q1)** bereitgestellt – Q1 ist TRUE, sobald der Zählerstand Null erreicht hat. Die Übung zeigt die direkte Verbindung von UDINT auf LREAL ohne manuelle Konvertierung und demonstriert die Kombination von Zählerlogik mit physischer Ausgabe.

Verwendete Funktionsbausteine (FBs)

Die Übung besteht aus einer SubApplikation (SubAppType), die fünf Funktionsbausteine enthält. Nachfolgend werden alle verwendeten FBs beschrieben.

Sub-Bausteine

1. **FB_CTD_UDINT** (IEC 61131-3 Rückwärtszähler)
   - **Typ**: `iec61131::counters::FB_CTD_UDINT`
   - **Parameter**:
     - `PV` = `UDINT#10` (Voreinstellwert)
   - **Ereigniseingänge**:
     - `REQ` – wird durch einen steigenden Impuls der Eingänge CD oder LD getriggert
   - **Ereignisausgänge**:
     - `CNF` – quittiert die Ausführung nach einem REQ
   - **Dateneingänge**:
     - `CD` – Zählimpuls (herunterzählen)
     - `LD` – Ladeimpuls (setzt Zähler auf PV)
   - **Datenausgänge**:
     - `Q` (BOOL) – TRUE wenn Zählerstand = 0
     - `CV` (UDINT) – aktueller Zählerstand

- **Funktionsweise**: Der Baustein realisiert einen flankengetriggerten Abwärtszähler. Bei einem positiven Impuls auf CD wird CV um 1 dekrementiert; bei LD wird CV auf den Wert in PV gesetzt. Der Ausgang Q wird TRUE, sobald CV den Wert 0 erreicht.

2. **Input_CD** (Digitaler Eingang I1)
   - **Typ**: `logiBUS::io::DI::logiBUS_IX`
   - **Parameter**:
     - `QI` = `TRUE` (Aktivierung)
     - `Input` = `Input_I1` (physischer Anschluss)
   - **Ereignisausgang**:
     - `IND` – wird bei einer steigenden Flanke am Eingang gesendet
   - **Datenausgang**:
     - `IN` (BOOL) – aktueller Zustand des Eingangs
   - **Funktionsweise**: Stellt den physischen digitalen Eingang I1 (z. B. Taster oder Sensor) im System bereit. Bei Änderung des Zustands wird das Ereignis IND ausgelöst.
3. **Input_LD** (Digitaler Eingang I2)
   - **Typ**: `logiBUS::io::DI::logiBUS_IX`
   - **Parameter**:
     - `QI` = `TRUE`
     - `Input` = `Input_I2`
   - **Ereignisausgang**:
     - `IND`
   - **Datenausgang**:
     - `IN` (BOOL)
   - **Funktionsweise**: Identisch zu Input_CD, jedoch an physischem Eingang I2 angeschlossen – dient als Ladeimpuls für den Zähler.
4. **Output_Q1** (Digitaler Ausgang Q1)
   - **Typ**: `logiBUS::io::DQ::logiBUS_QX`
   - **Parameter**:
     - `QI` = `TRUE` (Aktivierung)
     - `Output` = `Output_Q1` (physischer Ausgang)
   - **Ereigniseingang**:
     - `REQ` – triggert die Ausgabe des aktuellen Werts am Ausgang
   - **Dateneingang**:
     - `OUT` (BOOL) – Wert, der am Ausgang ausgegeben werden soll
   - **Funktionsweise**: Setzt den physischen digitalen Ausgang Q1 auf den Wert, der am Dateneingang OUT anliegt. Sobald der Zähler Q = TRUE liefert, wird Q1 aktiv.
5. **Q_NumericValue_PHYS_LREAL** (Terminal-Ausgabe)
   - **Typ**: `isobus::UT::Q::Q_NumericValue_PHYS_LREAL`
   - **Parameter**:
     - `stObj` = `OutputNumber_N3` (Referenz auf das Terminal-Ausgabeobjekt)
   - **Ereigniseingang**:
     - `REQ` – triggert die Ausgabe des aktuellen physikalischen Werts
   - **Dateneingang**:
     - `lrPhys` (LREAL) – der auszugebende physikalische Wert
   - **Funktionsweise**: Der Baustein übernimmt einen LREAL-Wert und gibt ihn über ein Terminal aus (z. B. auf einem Bedienpanel oder in einer Konsole). Der Zählerstand CV vom Typ UDINT wird ohne Konvertierung direkt an diesen Eingang angeschlossen.

Programmablauf und Verbindungen

Der Ablauf wird durch die Ereignis- und Datenverbindungen im SubAppNetwork gesteuert:

- **Ereignisverkettung**:
  - Eine steigende Flanke an `Input_CD` oder `Input_LD` (jeweils `IND`) löst den Ereigniseingang `REQ` des Zählers `FB_CTD_UDINT` aus.
  - Nach Verarbeitung quittiert der Zähler mit `CNF`. Dieses Ereignis wird an zwei Stellen weitergeleitet:

  - Zum einen an `Output_Q1.REQ` – damit der aktuelle boolesche Zustand (Q) am physischen Ausgang ausgegeben wird.
  - Zum anderen an `Q_NumericValue_PHYS_LREAL.REQ` – damit der aktuelle Zählerstand auf dem Terminal erscheint.

- **Datenverkettung**:
  - Der Datenausgang `Input_CD.IN` ist mit dem Zählereingang `FB_CTD_UDINT.CD` verbunden – das Signal des Tasters I1 wird als Zählimpuls verwendet.
  - `Input_LD.IN` ist mit `FB_CTD_UDINT.LD` verbunden – der Taster I2 lädt den Voreinstellwert.
  - Der Zählerausgang `Q` (BOOL) geht an den Dateneingang `OUT` von `Output_Q1`.
  - Der Zählerausgang `CV` (UDINT) geht an den Dateneingang `lrPhys` von `Q_NumericValue_PHYS_LREAL`.

Die beiden Kommentare im Netzwerk weisen darauf hin, dass alternativ auch der normale Ausgang (Q) verwendet werden könnte, und dass UDINT ohne explizite Konvertierung auf LREAL geschlossen werden darf (intern erfolgt eine implizite Typumwandlung).

Zusammenfassung

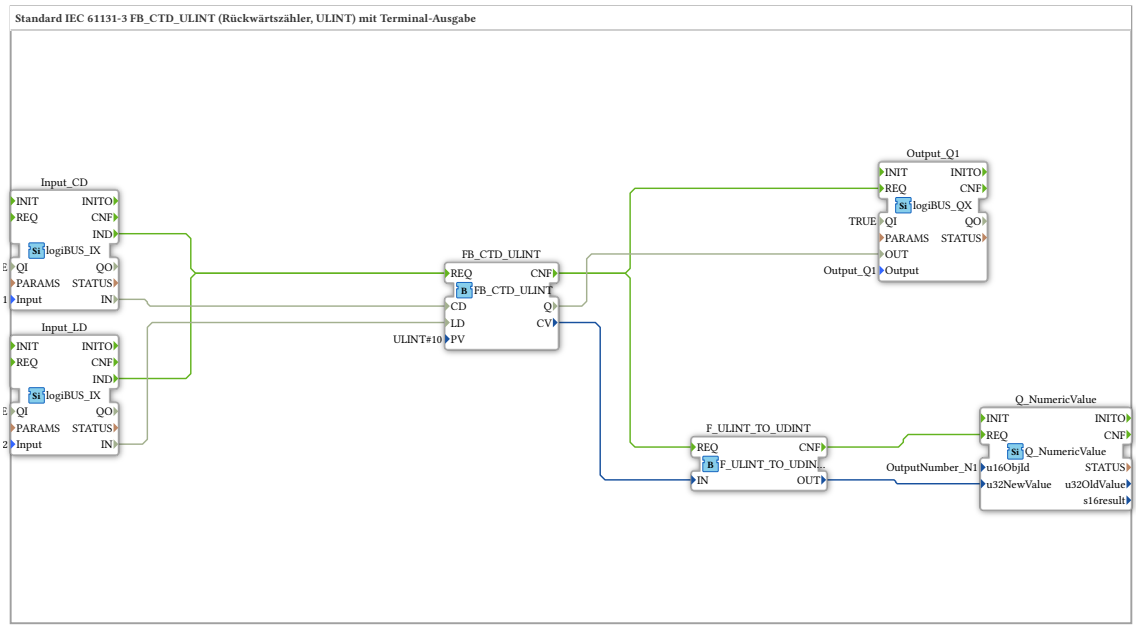
- **Lernziele**: Kennenlernen und Anwenden des IEC 61131-3 Rückwärtszählers (CTD) mit UDINT-Zählwerten, sowie die Integration von digitalen Ein-/Ausgängen und einer physischen Terminalausgabe.
- **Schwierigkeitsgrad**: Einfach bis mittel – geeignet zum Einstieg in die Zählerlogik und die Verwendung von physikalischen Ausgabeblöcken.
- **Vorkenntnisse**: Grundlegende Kenntnisse der 4diac-IDE und einfacher IEC-61131-3 Bausteine.
- **Bedienung**:
  1. Ordnen Sie die physischen Eingänge I1 (Taster für Herunterzählen) und I2 (Taster für Laden) an.
  2. Der Ausgang Q1 schaltet, sobald der Zählerstand Null erreicht.
  3. Das Terminal (OutputNumber_N3) zeigt den aktuellen Zählerstand als LREAL-Wert an.
  4. Starten Sie die Übung durch Auslösen einer steigenden Flanke an einem der Eingänge.

Die Übung demonstriert eine vollständige, praxisnahe Zähleranwendung mit sowohl digitaler als auch visualisierter Rückmeldung.

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

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

2.2.1.393 Uebung_219: Standard IEC 61131-3 FB_CTD_ULINT (Rückwärtszähler, ULINT) mit Terminal-Ausgabe



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung implementiert einen Rückwärtszähler (Abwärtszähler) nach IEC 61131-3 mit dem Funktionsbaustein `FB_CTD_ULINT` (Datentyp ULINT). Der Zähler wird über zwei digitale Eingänge gesteuert: **CD** (Count Down) dekrementiert den Zählerstand, **LD** (Load) lädt den voreingestellten Wert (PV). Der aktuelle Zählerstand wird auf einem Terminal (NumericValue) ausgegeben. Zusätzlich wird ein digitaler Ausgang gesetzt, wenn der Zählerstand null erreicht.

Verwendete Funktionsbausteine (FBs)

FB_CTD_ULINT (IEC 61131-3 Rückwärtszähler)

- **Typ**: iec61131::counters::FB_CTD_ULINT
- **Parameter**:
  - `PV` = ULINT#10 (Voreingestellter Startwert)
- **Ereigniseingänge/-ausgänge**:
  - `REQ` (Event-Eingang) – löst Zähleroperation aus
  - `CNF` (Event-Ausgang) – Bestätigung nach Ausführung
- **Dateneingänge/-ausgänge**:
  - `CD` (BOOL) – Abwärtszählimpuls
  - `LD` (BOOL) – Laden des PV
  - `Q` (BOOL) – Ausgang wird TRUE, wenn Zählerstand = 0
  - `CV` (ULINT) – Aktueller Zählerwert

Input_CD (Digitaler Eingang)

- **Typ**: logiBUS::io::DI::logiBUS_IX
- **Parameter**:
  - `QI` = TRUE (Qualifier)
  - `Input` = Input_I1 (Hardware-Eingang)
- **Ereignisausgang**: `IND` – wird bei Signaländerung ausgelöst
- **Datenausgang**: `IN` (BOOL) – aktueller Eingangswert

Input_LD (Digitaler Eingang)

- **Typ**: logiBUS::io::DI::logiBUS_IX
- **Parameter**:
  - `QI` = TRUE
  - `Input` = Input_I2
- **Ereignisausgang**: `IND`
- **Datenausgang**: `IN` (BOOL)

Output_Q1 (Digitaler Ausgang)

- **Typ**: logiBUS::io::DQ::logiBUS_QX
- **Parameter**:
  - `QI` = TRUE
  - `Output` = Output_Q1
- **Ereigniseingang**: `REQ` – übernimmt neuen Ausgangswert
- **Dateneingang**: `OUT` (BOOL) – zu setzender Ausgangswert

F_ULINT_TO_UDINT (Typkonvertierung)

- **Typ**: iec61131::conversion::F_ULINT_TO_UDINT
- **Ereigniseingang**: `REQ`
- **Ereignisausgang**: `CNF`
- **Dateneingang**: `IN` (ULINT)
- **Datenausgang**: `OUT` (UDINT)

Q_NumericValue (Terminal-Ausgabe)

- **Typ**: isobus::UT::Q::Q_NumericValue
- **Parameter**:
  - `u16ObjId` = OutputNumber_N1 (Kennung des Ausgabefeldes)
- **Ereigniseingang**: `REQ`
- **Dateneingang**: `u32NewValue` (UDINT)

Programmablauf und Verbindungen

Der Ablauf wird über Ereignisverbindungen gesteuert:

1. **Eingangssignale verarbeiten**:
   - Tritt eine Änderung an **Input_I1** (CD) oder **Input_I2** (LD) auf, so löst der entsprechende Eingangsbaustein (`Input_CD.IND` bzw. `Input_LD.IND`) ein Ereignis aus.
   - Beide Ereignisse sind mit dem **REQ**-Eingang des Zählers `FB_CTD_ULINT` verbunden. Das führt dazu, dass bei jedem der beiden Eingänge ein Zählvorgang ausgelöst wird.
2. **Zähleroperation**:
   - Der Zähler `FB_CTD_ULINT` führt abhängig vom Zustand der Datenleitungen aus:
     - Ist `LD` = TRUE, wird der Wert aus `PV` (ULINT#10) geladen.
     - Ist `CD` = TRUE (und `LD` = FALSE), wird der Zählerstand um 1 dekrementiert.
   - Nach Ausführung wird der **CNF**-Ereignisausgang aktiviert.
3. **Ausgang und Terminalausgabe**:
   - Das `CNF`-Ereignis wird an zwei Bausteine weitergeleitet:
     - **Output_Q1**: Der Datenausgang `Q` des Zählers (TRUE bei Zählerstand = 0) wird auf den Hardware-Ausgang `Output_Q1` gelegt.
     - **F_ULINT_TO_UDINT**: Der aktuelle Zählerwert `CV` (ULINT) wird in den Datentyp UDINT konvertiert, da die Terminalausgabe einen UDINT erwartet.
   - Nach der Konvertierung löst `F_ULINT_TO_UDINT.CNF` die **Q_NumericValue**-Ausgabe aus, sodass der Zählerwert auf dem Terminal (Objekt `OutputNumber_N1`) erscheint.

Hinweise aus den Kommentaren:

- Es wird empfohlen, einen **E_D_FF** (Edge-Detection-Flipflop) zwischen den Eingängen und dem Zähler einzufügen, um die Anzahl der Ereignisaufrufe zu reduzieren (z. B. bei schnellen Signaländerungen).
- Bei der Konvertierung `F_ULINT_TO_UDINT` kann es zu einem **Überlauf** kommen, da ULINT (64 Bit) in UDINT (32 Bit) umgewandelt wird. Der Wertebereich von ULINT ist größer; es werden nur die unteren 32 Bit übernommen. Dies ist bei der Wahl der Zählerwerte zu beachten.

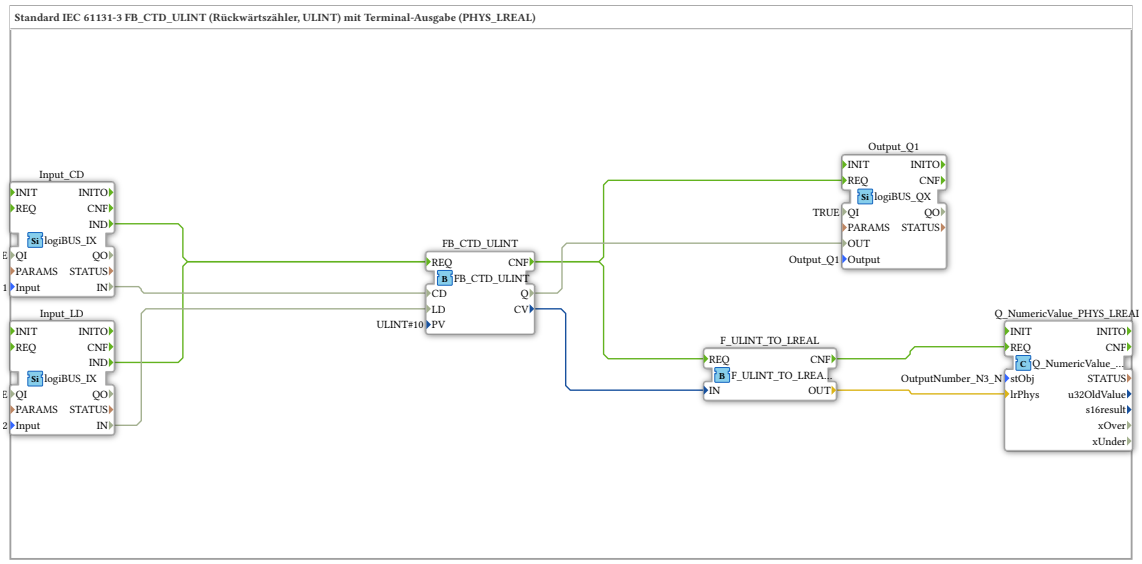
Zusammenfassung

Die Übung **Uebung_219** demonstriert einen IEC 61131-3 konformen Rückwärtszähler mit Terminalausgabe. Sie verbindet digitale Eingabe, Abwärtszählfunktion, Ausgabe auf einen digitalen Ausgang und numerische Anzeige auf einem Display. Dabei werden Typkonvertierung, Ereignisverkettung und Hardware-Schnittstellen trainiert. Die bereitgestellte Implementierung ist eine SubApp, die als wiederverwendbarer Baustein in der 4diac-IDE genutzt werden kann.

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

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

2.2.1.394 Uebung_219b: Standard IEC 61131-3 FB_CTD_ULINT (Rückwärtszähler, ULINT) mit Terminal-Ausgabe (PHYS_LREAL)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung demonstriert einen nach IEC 61131-3 standardisierten Rückwärtszähler (FB_CTD_ULINT) mit einem Zählbereich von ULINT (0 … 18.446.744.073.709.551.615). Der Zähler wird über zwei digitale Eingänge angesteuert: **CD** (Count Down) dekrementiert den aktuellen Zählwert bei jeder steigenden Flanke, **LD** (Load) setzt den Zählwert auf den vorgegebenen Preset-Wert (PV = 10) zurück. Der aktuelle Zählwert wird über eine Typkonvertierung in eine physikalische Gleitkommazahl (PHYS_LREAL) gewandelt und auf einem Terminal ausgegeben. Gleichzeitig wird ein digitaler Ausgang gesetzt, wenn der Zählwert den Wert 0 erreicht.

Verwendete Funktionsbausteine (FBs)

Bausteinname	Typ	Parameter / Einstellungen
FB_CTD_ULINT	`iec61131::counters::FB_CTD_ULINT`	`PV = ULINT#10`
Input_CD	`logiBUS::io::DI::logiBUS_IX`	`QI = TRUE`, `Input = Input_I1`
Input_LD	`logiBUS::io::DI::logiBUS_IX`	`QI = TRUE`, `Input = Input_I2`
Output_Q1	`logiBUS::io::DQ::logiBUS_QX`	`QI = TRUE`, `Output = Output_Q1`
F_ULINT_TO_LREAL	`iec61131::conversion::F_ULINT_TO_LREAL`	– (keine Parameter)
Q_NumericValue_PHYS_LREAL	`EBUS::UT::Q::Q_NumericValue_PHYS_LREAL`	`stObj = OutputNumber_N3`

Funktionsweise der einzelnen Bausteine:

Baustein	Beschreibung
FB_CTD_ULINT	Rückwärtszähler (CTD) für vorzeichenlose lange Ganzzahlen (ULINT). Beim Ereignis **REQ** wird abhängig vom aktuell aktiven Eingang (CD oder LD) entweder der Zählwert dekrementiert oder der Preset-Wert geladen. Der aktuelle Zählwert steht am Ausgang **CV**, der Nullstand am Ausgang **Q** zur Verfügung.
Input_CD	Digitaler Eingangsbaustein, der das physische Signal `Input_I1` (Taster/Schalter) einliest und bei einer positiven Flanke das Ereignis **IND** auslöst.

Input_LD	Digitaler Eingangsbaustein, der das physische Signal `Input_I2` einliest und bei einer positiven Flanke das Ereignis **IND** auslöst.
Output_Q1	Digitaler Ausgangsbaustein, der bei einem Ereignis **REQ** den Wert am Dateneingang **OUT** auf den physischen Ausgang `Output_Q1` übernimmt.
F_ULINT_TO_LREAL	Konvertierungsbaustein, der einen ULINT-Wert in den Typ LREAL (64-Bit Gleitkommazahl) umwandelt.
Q_NumericValue_PHYS_LREAL	Terminal-Ausgabebaustein für physikalische Gleitkommazahlen. Er gibt den übergebenen Wert am konfigurierten Objekt `OutputNumber_N3` aus.

Programmablauf und Verbindungen

Der Ablauf wird durch Ereignis- und Datenverbindungen gesteuert:

1. **Eingangssignale erfassen**
   - `Input_CD.IND` (steigende Flanke an `Input_I1`) wird mit `FB_CTD_ULINT.REQ` verbunden.
   - `Input_LD.IND` (steigende Flanke an `Input_I2`) wird ebenfalls mit `FB_CTD_ULINT.REQ` verbunden.

   ⟶ Der Zähler wird bei **jeder** positiven Flanke an einem der beiden Eingänge aktiviert. Die Unterscheidung, ob dekrementiert oder geladen wird, erfolgt über die Datenverbindungen.
2. **Datenwerte zuweisen**
   - `Input_CD.IN` ⟶ `FB_CTD_ULINT.CD` (Count Down)
   - `Input_LD.IN` ⟶ `FB_CTD_ULINT.LD` (Load)

   ⟶ Die logischen Zustände der beiden Eingänge bestimmen die Aktion:
   - Ist **CD** = **TRUE** und **LD** = **FALSE**, wird der Zählwert dekrementiert.
   - Ist **LD** = **TRUE** (unabhängig von CD), wird der Preset-Wert (10) geladen.
3. **Ausgabe nach Verarbeitung** Nach Abschluss der Zähleroperation wird das Ereignis **CNF** des Zählers ausgelöst. Dieses ist mit zwei nachfolgenden Bausteinen verbunden:
   - `Output_Q1.REQ`: Der aktuelle Zustand von `FB_CTD_ULINT.Q` (Zählerstand = 0 ⟶ TRUE) wird auf den digitalen Ausgang `Output_Q1` geschrieben.
   - `F_ULINT_TO_LREAL.REQ`: Der aktuelle Zählwert (`FB_CTD_ULINT.CV`) wird in eine LREAL-Zahl umgewandelt.
4. **Terminal-Ausgabe** Nach der Konvertierung triggert `F_ULINT_TO_LREAL.CNF` den Baustein `Q_NumericValue_PHYS_LREAL.REQ`. Der konvertierte Wert (`F_ULINT_TO_LREAL.OUT`) wird als physikalische Gleitkommazahl am Terminal unter `OutputNumber_N3` angezeigt.

Die gesamte Logik arbeitet **ereignisgesteuert**: Jede Änderung an einem der Eingänge löst eine vollständige Verarbeitungskette aus – vom Einlesen über die Zählerlogik bis zur Ausgabe des aktuellen Zählwerts und des digitalen Signals.

Zusammenfassung

Die Übung **Uebung_219b** setzt einen IEC 61131-3-konformen Rückwärtszähler (FB_CTD_ULINT) in der 4diac-IDE um. Sie zeigt:

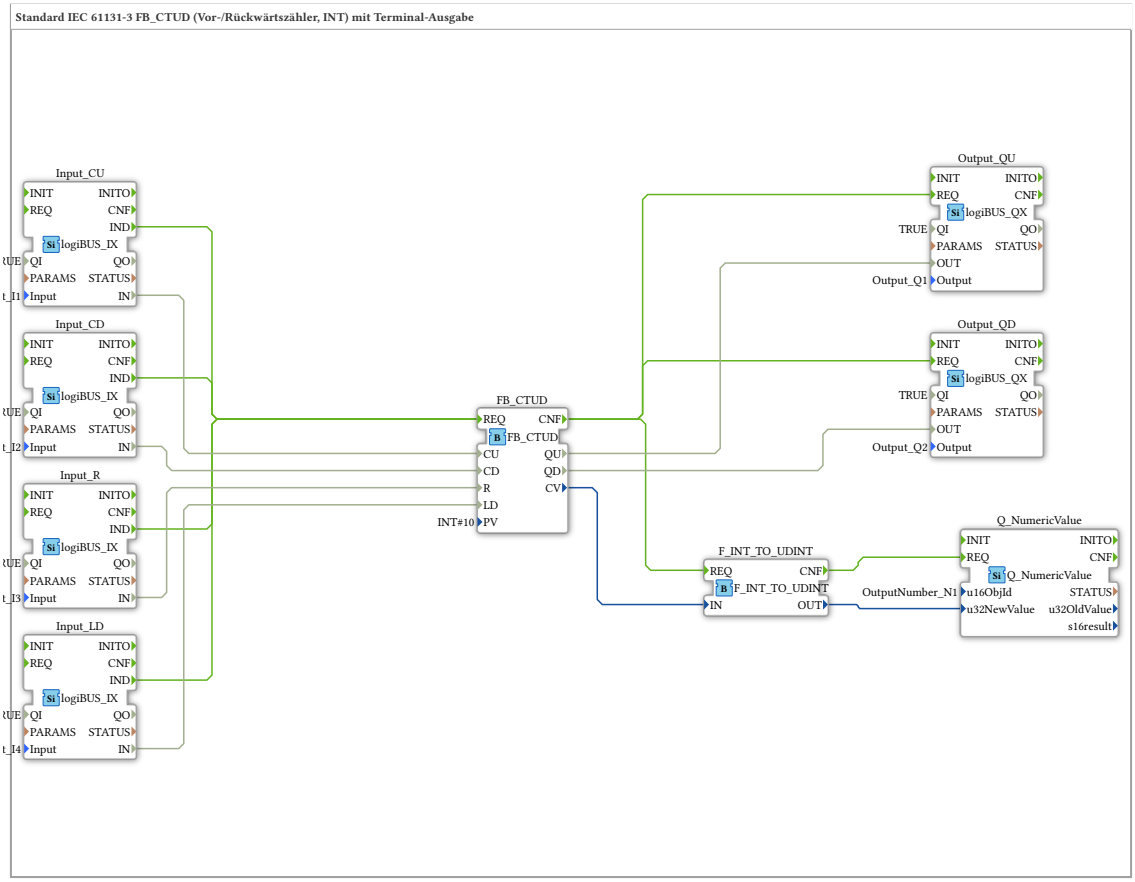
- die Parametrierung eines standardisierten Zählbausteins (Preset = 10),
- die Anbindung digitaler Ein-/Ausgänge über logiBUS-Bausteine,
- die Typkonvertierung von ULINT nach LREAL,
- die Ausgabe von Messwerten auf einem Terminal (PHYS_LREAL).

Der Schwerpunkt liegt auf dem Verständnis von Ereignis- und Datenflüssen sowie der strukturierten Verschaltung von Funktionsbausteinen in einer Automatisierungsanwendung.

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

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

2.2.1.395 Uebung_220: Standard IEC 61131-3 FB_CTUD (Vor-/Rückwärtszähler, INT) mit Terminal-Ausgabe



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung realisiert einen kombinierten Vor-/Rückwärtszähler (Up/Down Counter) nach IEC 61131-3 (FB CTUD) mit einer Preset-Schwelle von 10. Die Zählerstände werden sowohl über digitale Ausgänge als auch über einen numerischen Wert auf einem Terminal ausgegeben. Die Eingänge werden über logiBUS-Eingangsbausteine bereitgestellt.

Verwendete Funktionsbausteine (FBs)

- **FB_CTUD** (Typ: `iec61131::counters::FB_CTUD`): Standard IEC 61131-3 Vor-/Rückwärtszähler.
  - Parameter: `PV` = `INT#10` (Presetwert)
  - Ereigniseingänge: `REQ` (Anforderung)
  - Ereignisausgänge: `CNF` (Bestätigung)
  - Dateneingänge: `CU` (Count Up), `CD` (Count Down), `R` (Reset), `LD` (Load)
  - Datenausgänge: `QU` (Ausgang bei Erreichen Preset), `QD` (Ausgang bei Erreichen 0), `CV` (aktueller Zählerwert)
- **Eingangsbausteine** (Typ: `logiBUS_IX`):
  - `Input_CU` (Eingang für Zählimpulse Aufwärts) – verbunden mit `Input_I1`
  - `Input_CD` (Eingang für Zählimpulse Abwärts) – verbunden mit `Input_I2`
  - `Input_R` (Reset-Eingang) – verbunden mit `Input_I3`
  - `Input_LD` (Load-Eingang) – verbunden mit `Input_I4`
  - Alle haben den Parameter `QI` = `TRUE`.
- **Ausgangsbausteine** (Typ: `logiBUS_QX`):
  - `Output_QU` (Ausgang QU) – verbunden mit `Output_Q1`
  - `Output_QD` (Ausgang QD) – verbunden mit `Output_Q2`

  - Alle haben den Parameter `QI` = `TRUE`.
- **F_INT_TO_UDINT** (Typ: `iec61131::conversion::F_INT_TO_UDINT`): Konvertiert den Integer-Zählerwert (CV) in einen vorzeichenlosen Doppel-Integer (UDINT) für die Terminalausgabe.
- **Q_NumericValue** (Typ: `isobus::UT::Q::Q_NumericValue`): Gibt einen numerischen Wert auf dem Terminal aus.
  - Parameter: `u16ObjId` = `OutputNumber_N1`

Hinweis: Die Verwendung von `F_INT_TO_UDINT` ist laut Kommentar *„großer Quatsch … keine Negativen Zahlen möglich“*, da der Zählerwert auch negativ werden kann (wenn mehr Rückwärts- als Vorwärtsimpulse erfolgen). In dieser Übung wird jedoch der Umgang mit dem FB CTUD demonstriert.

Programmablauf und Verbindungen

Die Übung zeigt die typische Anwendung eines IEC 61131-3 Zählers mit Hardware-Ein- und Ausgängen über den logiBUS.

- **Ereignissteuerung**: Jeder Eingangsbaustein (`Input_CU`, `Input_CD`, `Input_R`, `Input_LD`) löst bei einer Flanke (`IND`) eine `REQ` an den `FB_CTUD` aus. Somit wird der Zähler bei jedem Impuls aktualisiert. Nach der Verarbeitung gibt der `FB_CTUD` eine `CNF`, die gleichzeitig die Ausgangsbausteine (`Output_QU`, `Output_QD`) sowie die Konvertierung und Terminalausgabe triggert.
- **Datenverbindungen**:
  - Die digitalen Eingangssignale (`IN` der `logiBUS_IX`) werden direkt an die entsprechenden Dateneingänge des Zählers (`CU`, `CD`, `R`, `LD`) geführt.
  - Die Zählerausgänge `QU` und `QD` werden an die logiBUS-Ausgänge (`OUT` von `logiBUS_QX`) angeschlossen.
  - Der aktuelle Zählerwert `CV` wird über `F_INT_TO_UDINT` in `UDINT` konvertiert und an `Q_NumericValue` übergeben, der diesen auf dem Terminal (Ausgabenummer `N1`) anzeigt.
- **Lernziele**:
  - Verständnis des IEC 61131-3 FB_CTUD (Vor-/Rückwärtszähler) mit allen Funktionen.
  - Einbindung von Hardware-Ein-/Ausgängen über logiBUS.
  - Ausgabe eines numerischen Wertes auf einem Terminal.
  - Erkennen von Einschränkungen bei der Datentypkonvertierung (Negativwerte).
- **Schwierigkeitsgrad**: Mittel. Vorkenntnisse in IEC 61131-3 und der 4diac-IDE sind hilfreich.

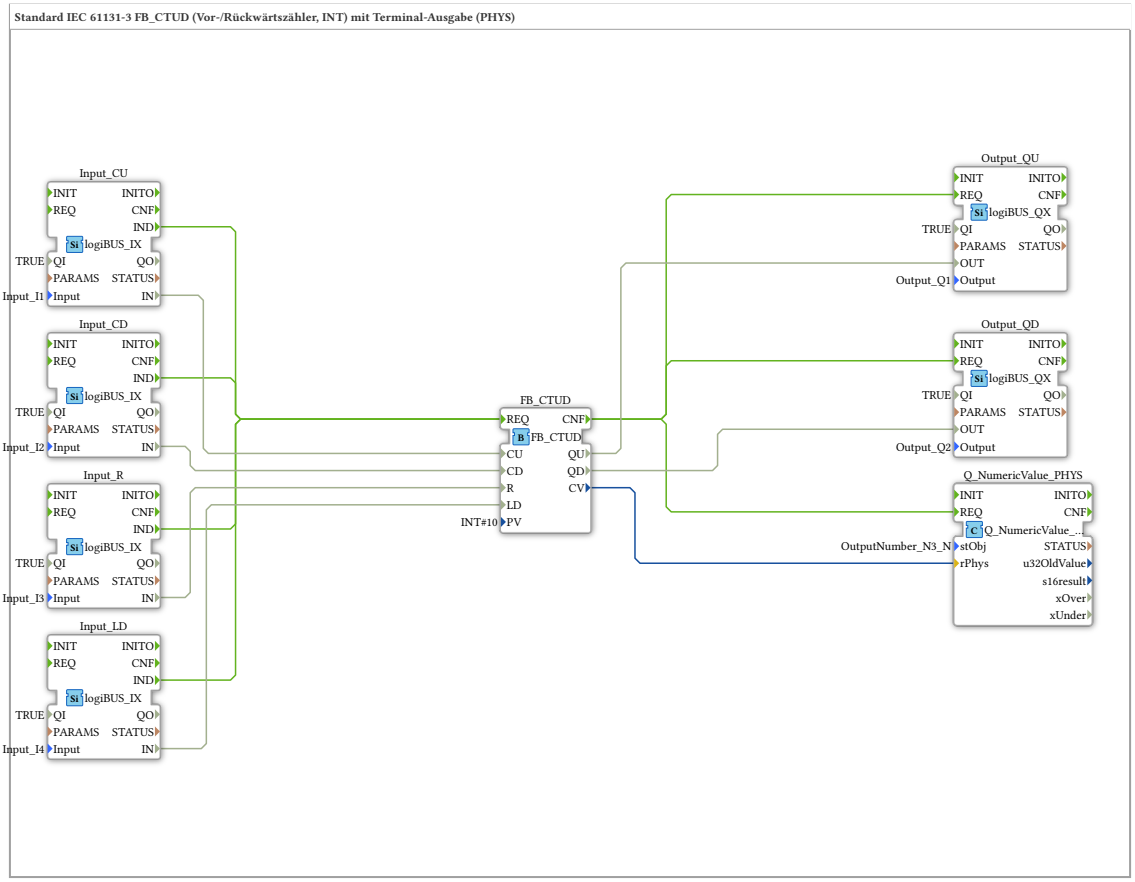
Zusammenfassung

In dieser Übung wurde ein Vor-/Rückwärtszähler nach IEC 61131-3 implementiert. Der Zähler zählt bei jeder steigenden Flanke an den Eingängen `CU` (Aufwärts) und `CD` (Abwärts). Ein Reset (`R`) setzt den Zähler auf `0` zurück, Load (`LD`) lädt den Presetwert `PV`. Die Ausgänge `QU` und `QD` zeigen an, ob der Zählerstand den Presetwert erreicht (`QU`) bzw. `0` (`QD`). Zusätzlich wird der aktuelle Zählerwert auf einem Terminal ausgegeben, wobei die Konvertierung in `UDINT` für positive Werte funktioniert. Die Übung demonstriert die vollständige Integration von Standard-FBs mit logiBUS-Hardware und Terminalausgabe in der 4diac-IDE.

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

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

2.2.1.396 Uebung_220b: Standard IEC 61131-3 FB_CTUD (Vor-/Rückwärtszähler, INT) mit Terminal-Ausgabe (PHYS)



• – ○ • – ○ • – ○ •

Einleitung

In dieser Übung wird ein Vor-/Rückwärtszähler (Typ CTUD) nach IEC 61131-3 realisiert. Der Zähler besitzt einen Vorwärtszähleingang (CU), einen Rückwärtszähleingang (CD), einen Rücksetzeingang (R) und einen Ladeeingang (LD). Der aktuelle Zählwert (CV) wird auf einem physikalischen Terminal (PHYS) ausgegeben. Zusätzlich werden die beiden Zählerendstände QU (Überlauf bei Erreichen des Maximalwerts) und QD (Unterlauf bei Erreichen des Minimalwerts) auf digitale Ausgänge geführt. Die Steuerung erfolgt über vier digitale Eingänge (I1 … I4).

Verwendete Funktionsbausteine (FBs)

Die Übung verwendet keine weiteren Sub-Bausteine, sondern setzt direkt aus dem Netzwerk folgende vordefinierte Funktionsbausteine ein:

- **FB_CTUD** (Typ: `iec61131::counters::FB_CTUD`)
  Parameter: `PV = INT#10` (Zählschwellwert, entspricht einem Vorwahlwert von 10).
  Der Baustein arbeitet als Vor-/Rückwärtszähler mit Ereignissteuerung und liefert die Ausgänge `QU`, `QD` und `CV`.
- **Input_CU**, **Input_CD**, **Input_R**, **Input_LD** (Typ: `logiBUS::io::DI::logiBUS_IX`)
  Parameter: `QI = TRUE`, `Input = Input_I1` … `Input_I4`.
  Vier digitale Eingänge, die die physikalischen Eingänge I1 bis I4 auf die internen Datenwerte abbilden.
- **Output_QU**, **Output_QD** (Typ: `logiBUS::io::DQ::logiBUS_QX`)
  Parameter: `QI = TRUE`, `Output = Output_Q1` bzw. `Output_Q2`.
  Zwei digitale Ausgänge, die die Überlauf- und Unterlaufbedingungen des Zählers an die physikalischen Ausgänge Q1 und Q2 weitergeben.

- **Q_NumericValue_PHYS** (Typ: `isobus::UT::Q::Q_NumericValue_PHYS`)
  Parameter: `stObj = OutputNumber_N3` (Vorgabe aus dem Pool `DefaultPool_Numeric`).
  Dieser Baustein dient der Ausgabe des aktuellen Zählwerts als Realwert auf einem Terminal. Er erhält den Zahlenwert über den Dateneingang `rPhys`.

Programmablauf und Verbindungen

Der Ablauf wird über Ereignis- und Datenverbindungen gesteuert.

Ereignisverbindungen

- Die `IND`-Ereignisse aller vier digitalen Eingänge (Input_CU, Input_CD, Input_R, Input_LD) sind auf den `REQ`-Eingang des Zählers FB_CTUD geschaltet. Jede steigende Flanke an einem der Eingänge löst somit eine Neuberechnung des Zählers aus.
- Nach erfolgreicher Berechnung sendet der Zähler das `CNF`-Ereignis. Dieses wird auf die `REQ`-Eingänge der drei Ausgabebausteine (Output_QU, Output_QD, Q_NumericValue_PHYS) verteilt, sodass die Ausgangswerte parallel aktualisiert werden.

Datenverbindungen

- Die digitalen Eingangswerte (`IN`) von Input_CU, Input_CD, Input_R und Input_LD werden auf die entsprechenden Dateneingänge des Zählers geführt (`CU`, `CD`, `R`, `LD`).
- Vom Zähler werden die binären Ausgangssignale `QU` und `QD` auf die Dateneingänge `OUT` der Ausgangsbausteine Output_QU und Output_QD übertragen.
- Der aktuelle Zählwert `CV` (Datentyp `INT`) wird auf den Eingang `rPhys` des Terminal-Ausgabebausteins gelegt. Ein Kommentar im Netzwerk erklärt, dass `INT` ohne explizite Konvertierung auf `REAL` geschlossen werden kann (implizite Typumwandlung ist erlaubt).

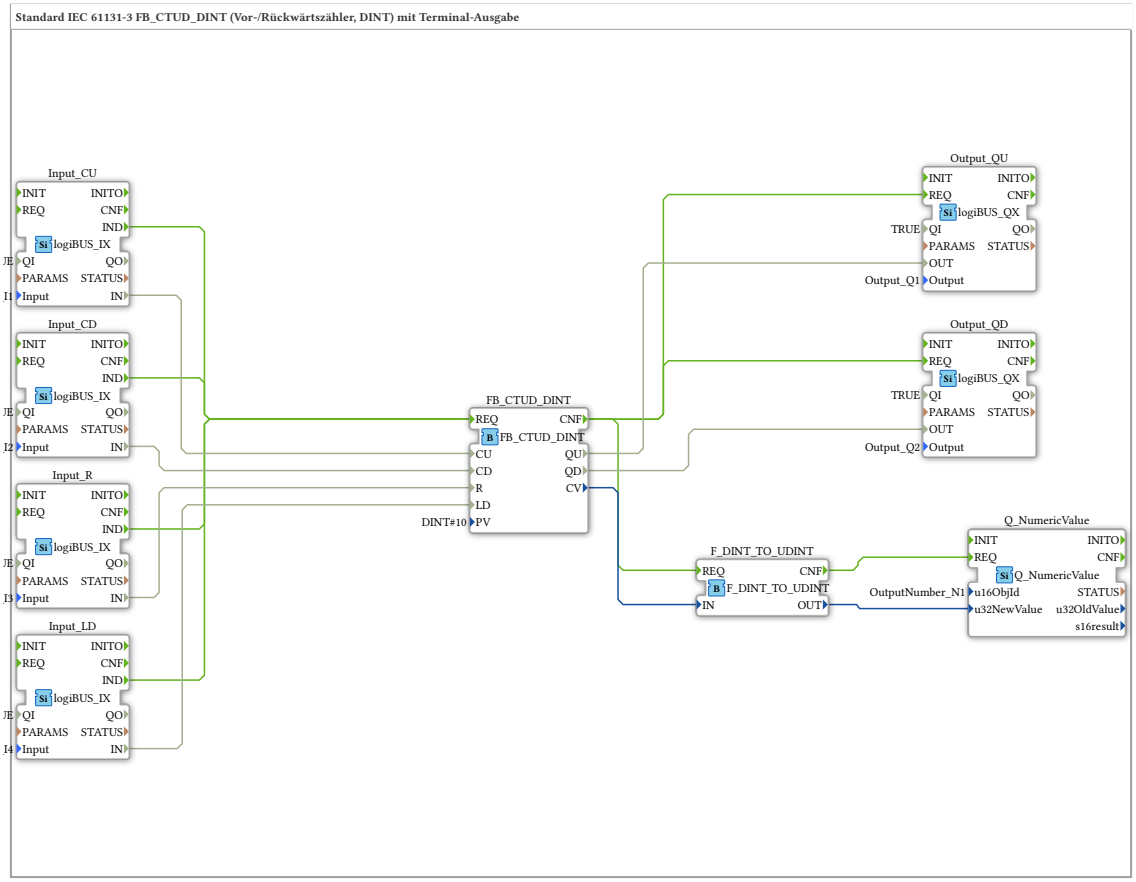
Besonderheiten

- Der Zähler arbeitet nach der CTUD-Spezifikation: Ein positiver Impuls an `CU` erhöht den Zählwert, an `CD` verringert ihn. Ein Signal an `R` setzt den Zählwert auf Null, ein Signal an `LD` lädt den in `PV` hinterlegten Vorwahlwert.
- Die Übung vermittelt das Verständnis eines kombinierten Vor-/Rückwärtszählers mit Hardware-Anbindung und visueller Terminalausgabe. Vorkenntnisse zu IEC 61131-3 Zählern und grundlegenden Ereignis-/Datenverbindungen sind hilfreich.

Zusammenfassung

Die Übung „Uebung_220b“ realisiert einen vollständigen IEC-61131-3-konformen Vor-/Rückwärtszähler (CTUD) mit digitalen Ein- und Ausgängen sowie einer physikalischen Terminalausgabe des aktuellen Zählwerts. Die Steuerung erfolgt über vier Taster, die Zählerstände werden als Überlauf bzw. Unterlauf auf zwei Ausgänge gegeben. Der Ablauf demonstriert die typische Verschaltung eines Ereignis-gesteuerten Zählers mit dezentralen I/O-Modulen und einem numerischen Anzeigebaustein.

2.2.1.397 Uebung_221: Standard IEC 61131-3 FB_CTUD_DINT (Vor-/Rückwärtszähler, DINT) mit Terminal-Ausgabe



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung implementiert einen kombinierten Vor-/Rückwärtszähler (engl. Up/Down Counter) basierend auf dem Standard IEC 61131-3 Funktionsbaustein `FB_CTUD_DINT`. Der gezählte Wert wird als Ganzzahl (DINT) geführt und über eine Konvertierung auf einer numerischen Anzeige (Terminal) ausgegeben. Zusätzlich werden zwei binäre Ausgänge gesetzt, die anzeigen, ob der Zähler den oberen (QU) oder unteren (QD) Grenzwert erreicht hat.

Die Steuerung erfolgt über vier digitale Eingänge (CU, CD, Reset, Laden des Anfangswerts), die über den logiBUS angeschlossen sind.

Verwendete Funktionsbausteine (FBs)

Kern-Funktionsbaustein

- **FB_CTUD_DINT** (Typ: `iec61131::counters::FB_CTUD_DINT`)
  - Parametrierung: `PV` = `DINT#10` (Vergleichswert für QU/QD)
  - Funktion: IEC 61131-3 konformer Vor-/Rückwärtszähler mit DINT-Werten. Zählt bei jedem positiven Flanke an CU (Count Up) oder CD (Count Down). Setzt QU auf TRUE, wenn `CV >= PV`, und QD auf TRUE, wenn `CV <= 0`. Über `R` wird der Zähler auf 0 zurückgesetzt; über `LD` wird der aktuelle Wert von `PV` geladen.

Digitale Eingänge (logiBUS)

- **Input_CU**, **Input_CD**, **Input_R**, **Input_LD** (Typ: `logiBUS::io::DI::logiBUS_IX`)
  - Parametrierung: `QI` = `TRUE`
  - Funktion: Lesen der physikalischen Eingänge `Input_I1` bis `Input_I4` und Bereitstellung des Binärwerts am Datenausgang `IN`. Bei jeder Änderung wird ein Ereignis `IND` ausgelöst.

Digitale Ausgänge (logiBUS)

- **Output_QU**, **Output_QD** (Typ: `logiBUS::io::DQ::logiBUS_QX`)
  - Parametrierung: `QI` = `TRUE`
  - Funktion: Setzen der physikalischen Ausgänge `Output_Q1` und `Output_Q2` entsprechend des anliegenden Werts am Dateneingang `OUT`.

Ausgabe des Zählerwerts auf Terminal

- **Q_NumericValue** (Typ: `isobus::UT::Q::Q_NumericValue`)
  - Parametrierung: `u16ObjId` = `OutputNumber_N1`
  - Funktion: Zeigt einen numerischen Wert (UDINT) auf einem Terminal (z. B. HMI) an. Der Datenwert wird über `u32NewValue` übergeben.
- **F_DINT_TO_UDINT** (Typ: `iec61131::conversion::F_DINT_TO_UDINT`)
  - Funktion: Konvertiert einen DINT-Wert in einen UDINT-Wert. Dadurch wird der Zählerstand (der negativ werden könnte) als vorzeichenlose Ganzzahl dargestellt. (Hinweis: Diese Konvertierung ist nicht sinnvoll für negative Zählerstände, da negative DINT-Werte in UDINT umgedeutet werden.)

Programmablauf und Verbindungen

1. **Ereignisverknüpfung**: Jeder der vier digitalen Eingänge (Input_CU, Input_CD, Input_R, Input_LD) löst bei einer Zustandsänderung (Ereignis `IND`) die Ausführung des Zählers `FB_CTUD_DINT` aus (Eingangsereignis `REQ`). → Alle Eingänge sind direkt mit demselben `REQ`-Ereignis des Zählers verbunden. Dies bewirkt, dass bei jeder Änderung eines beliebigen Eingangs der Zähler neu verarbeitet wird.
2. **Datenverknüpfung**:
   - Die Eingangswerte `IN` der digitalen Eingänge werden auf die entsprechenden Dateneingänge des Zählers geführt: `CU`, `CD`, `R`, `LD`.
   - Nach der Zählerverarbeitung (`CNF`-Ereignis) werden:
     - Die Ausgänge `QU` und `QD` an die Ausgangsbausteine `Output_QU` und `Output_QD` weitergeleitet. Diese setzen die physikalischen Ausgänge.
     - Der aktuelle Zählerstand `CV` wird über `F_DINT_TO_UDINT` in einen vorzeichenlosen Wert umgewandelt und an `Q_NumericValue` übergeben, der den Wert auf dem Terminal anzeigt.
3. **Ablauf**:
   - Ein positiver Flanke an CU erhöht den Zähler um 1.
   - Ein positiver Flanke an CD verringert den Zähler um 1.
   - Ein positiver Flanke an R setzt den Zähler auf 0.
   - Ein positiver Flanke an LD lädt den Wert von `PV` (hier 10) in den Zähler.
   - Sobald der Zählerstand den Vergleichswert `PV` erreicht oder überschreitet, wird `QU` auf TRUE gesetzt; bei Unterschreiten von 0 wird `QD` auf TRUE gesetzt.

Lernziele:

- Verständnis und Anwendung des IEC 61131-3 Standardzählers `CTUD`.
- Einbindung von logiBUS-I/O-Modulen in eine 4diac-Applikation.
- Ausgabe numerischer Werte auf einem Terminal.
- Umgang mit Ereignis- und Datenverbindungen in 4diac.

Schwierigkeitsgrad: Einfach **Vorkenntnisse**: Grundlagen der 4diac-IDE, IEC 61131-3 Bausteine, logiBUS-Konfiguration.

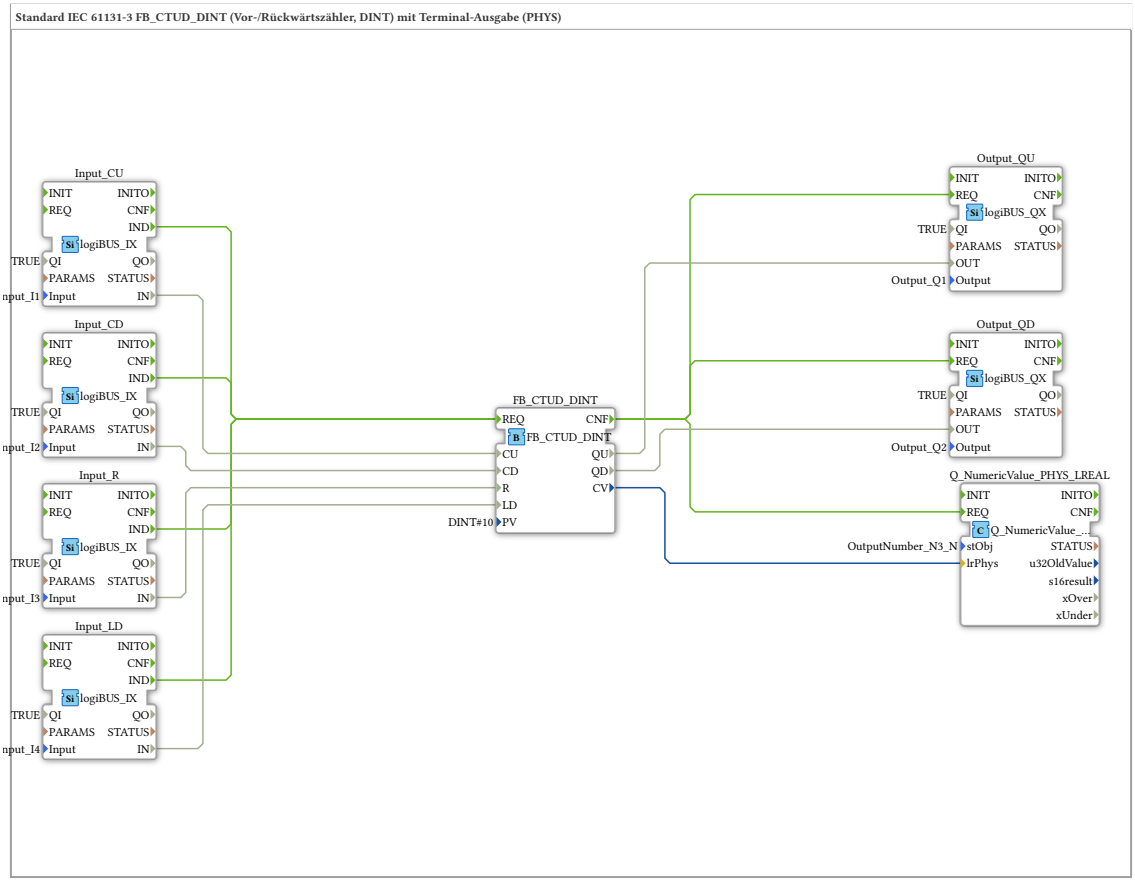
Zusammenfassung

Die Übung **Uebung_221** realisiert einen voll funktionsfähigen Vor-/Rückwärtszähler nach IEC 61131-3 mit vier digitalen Eingängen, zwei digitalen Ausgängen und einer numerischen Terminalanzeige. Der Aufbau demonstriert die saubere Trennung von Ereignis- und Datenflüssen sowie die Integration von Standardbibliotheken (IEC 61131-3, logiBUS, isobus) in einer 4diac-Subapplikation. Die Konvertierung von DINT nach UDINT ist ein bewusst gesetzter Hinweis auf mögliche Fallstricke bei der Darstellung negativer Zahlen.

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

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

2.2.1.398 Uebung_221b: Standard IEC 61131-3 FB_CTUD_DINT (Vor-/Rückwärtszähler, DINT) mit Terminal-Ausgabe (PHYS)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung implementiert einen kombinierten Vor-/Rückwärtszähler nach IEC 61131-3 (Typ `FB_CTUD_DINT`) und gibt den aktuellen Zählerstand sowie die Zählerstatus (Überlauf/Unterlauf) über digitale Ausgänge und eine Terminalausgabe (PHYS) aus. Der Zähler kann über vier digitale Eingänge gesteuert werden: Vorwärtszählen (CU), Rückwärtszählen (CD), Rücksetzen (R) und Laden des Startwertes (LD).

Verwendete Funktionsbausteine (FBs)

- **`FB_CTUD_DINT`** (Typ: `iec61131::counters::FB_CTUD_DINT`)
  - Parameter: `PV` = `DINT#10` (Startwert beim Laden)
  - Ereigniseingänge: `REQ` (Start der Verarbeitung)
  - Ereignisausgänge: `CNF` (Verarbeitung abgeschlossen)
  - Dateneingänge: `CU` (Count Up), `CD` (Count Down), `R` (Reset), `LD` (Load)
  - Datenausgänge: `QU` (Count Up erreicht), `QD` (Count Down erreicht), `CV` (aktueller Zählerwert)
- **Eingangs-FBs (logiBUS Digitaleingänge)**
  - `Input_CU` (Typ `logiBUS::io::DI::logiBUS_IX`): Eingangssignal für `CU`
  - `Input_CD` (Typ `logiBUS::io::DI::logiBUS_IX`): Eingangssignal für `CD`
  - `Input_R` (Typ `logiBUS::io::DI::logiBUS_IX`): Eingangssignal für `R`
  - `Input_LD` (Typ `logiBUS::io::DI::logiBUS_IX`): Eingangssignal für `LD`
  - Parameter aller: `QI` = `TRUE` (Aktivierung des Kanals), `Input` = zugeordneter logiBUS-Kanal (z.B. `Input_I1`..`Input_I4`)
- **Ausgangs-FBs (logiBUS Digitalausgänge)**
  - `Output_QU` (Typ `logiBUS::io::DQ::logiBUS_QX`): signalisiert `QU` (Zählerstand ≥ PV)

  - `Output_QD` (Typ `logiBUS::io::DQ::logiBUS_QX`): signalisiert `QD` (Zählerstand ≤ 0)
  - Parameter aller: `QI` = `TRUE`, `Output` = zugeordneter logiBUS-Kanal (z.B. `Output_Q1`, `Output_Q2`)
- **Terminalausgabe-FB**
  - `Q_NumericValue_PHYS_LREAL` (Typ `isobus::UT::Q::Q_NumericValue_PHYS_LREAL`): gibt den aktuellen Zählerstand (als LREAL) auf einem physikalischen Display aus
  - Parameter: `stObj` = `OutputNumber_N3` (Referenz auf das Terminal-Ausgabeelement)

Programmablauf und Verbindungen

Das System arbeitet ereignisgesteuert:

1. **Eingangsverarbeitung**: Jeder der vier Eingangs-FBs (Input_CU, Input_CD, Input_R, Input_LD) erzeugt bei einer Signaländerung ein Ereignis (`IND`).
2. **Zählerberechnung**: Alle vier Ereignisse sind mit dem `REQ`-Eingang des Zählers `FB_CTUD_DINT` verbunden. Dadurch wird der Zähler bei jedem neuen Eingangssignal ausgewertet.
3. **Ausgangsaktualisierung**: Nach Abschluss der Zählerberechnung (`CNF`) werden die Ausgangs-FBs und der Terminal-FB zeitgleich getriggert:
   - `Output_QU` erhält den Wert von `QU`
   - `Output_QD` erhält den Wert von `QD`
   - `Q_NumericValue_PHYS_LREAL` erhält den aktuellen Zählerstand `CV`

Datenverbindungen:

- `Input_CU.IN` ⟶ `FB_CTUD_DINT.CU`
- `Input_CD.IN` ⟶ `FB_CTUD_DINT.CD`
- `Input_R.IN` ⟶ `FB_CTUD_DINT.R`
- `Input_LD.IN` ⟶ `FB_CTUD_DINT.LD`
- `FB_CTUD_DINT.QU` ⟶ `Output_QU.OUT`
- `FB_CTUD_DINT.QD` ⟶ `Output_QD.OUT`
- `FB_CTUD_DINT.CV` ⟶ `Q_NumericValue_PHYS_LREAL.lrPhys`

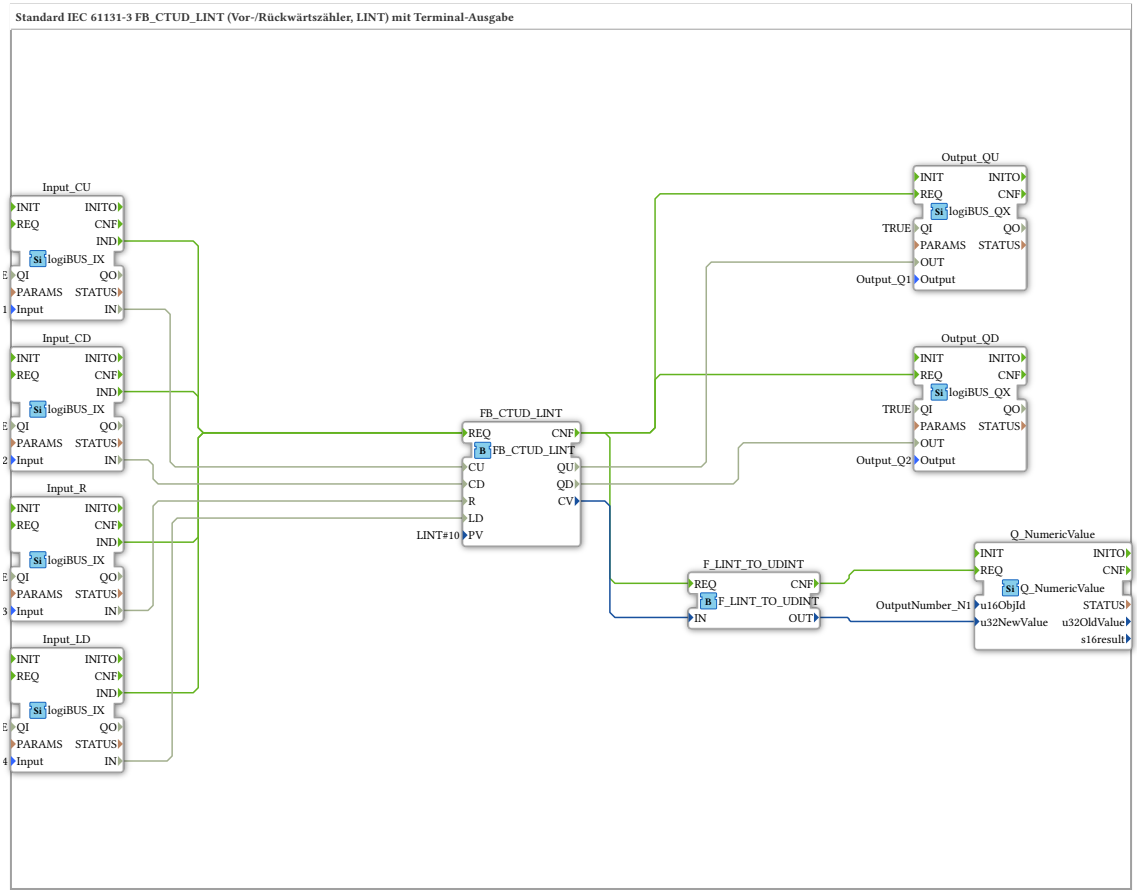
Verhalten des Zählers:

- Bei einer positiven Flanke an `CU` wird der Zähler um 1 erhöht.
- Bei einer positiven Flanke an `CD` wird der Zähler um 1 verringert.
- Bei einer positiven Flanke an `R` wird der Zähler auf 0 gesetzt.
- Bei einer positiven Flanke an `LD` wird der Zähler auf den Wert von `PV` (hier 10) gesetzt.
- Der Ausgang `QU` wird `TRUE`, sobald der Zählerstand ≥ PV ist; `QD` wird `TRUE`, sobald der Zählerstand ≤ 0 ist.

Zusammenfassung

Die Übung zeigt den Einsatz eines universellen IEC 61131-3 Vor-/Rückwärtszählers (`FB_CTUD_DINT`) in Kombination mit digitalen Ein- und Ausgängen sowie einer physikalischen Terminalausgabe. Der Zähler wird über vier Taster gesteuert, die Statusausgänge werden auf LEDs ausgegeben, und der aktuelle Zählerstand erscheint auf einem Display. Dies ist eine grundlegende Aufgabe zur Einführung in Zählfunktionen und die Signalverarbeitung in der Automatisierungstechnik.

2.2.1.399 Uebung_222: Standard IEC 61131-3 FB_CTUD_LINT (Vor-/Rückwärtszähler, LINT) mit Terminal-Ausgabe



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung implementiert einen Vor-/Rückwärtszähler gemäß IEC 61131-3 (Typ `FB_CTUD_LINT`) mit einem Wertebereich vom Typ `LINT`. Die Zählerstände werden über eine logiBUS-I/O-Kopplung an digitale Eingänge (Taster) gesteuert und an digitale Ausgänge (Signalleuchten) ausgegeben. Zusätzlich wird der aktuelle Zählerstand über einen numerischen Terminalausgang (`Q_NumericValue`) dargestellt. Die Übung demonstriert den Einsatz eines komplexen Zählers sowie die Umwandlung von `LINT` nach `UDINT` für die Ausgabe, wobei ein Kommentar auf die Einschränkung der Umwandlung hinweist (keine negativen Werte darstellbar).

Verwendete Funktionsbausteine (FBs)

Die Übung enthält folgende Funktionsbausteine direkt auf der obersten Ebene (keine Sub-Bausteine):

- **FB_CTUD_LINT** (Typ: `iec61131::counters::FB_CTUD_LINT`)
  - Parameter: `PV` = `LINT#10` (Voreinstellwert 10)
  - Ereigniseingänge: `REQ` (Anforderung)
  - Ereignisausgänge: `CNF` (Bestätigung)
  - Dateneingänge: `CU` (Aufwärtszählimpuls), `CD` (Abwärtszählimpuls), `R` (Rücksetzen auf 0), `LD` (Laden von PV)
  - Datenausgänge: `QU` (Aufwärtszähler erreicht PV), `QD` (Abwärtszähler erreicht 0), `CV` (aktueller Zählerstand)
  - **Funktionsweise**: Der Baustein arbeitet als vorwärts/rückwärts zählender Timer mit 64-Bit-Zähler (`LINT`). Bei jedem positiven Flanke an `CU` wird der Zähler erhöht, bei `CD` verringert. Über `R` wird auf `0` gesetzt, über `LD` auf den Wert von `PV` geladen. `QU` ist aktiv, wenn `CV >= PV`, `QD` aktiv, wenn `CV <= 0`.
- **Input_CU** (Typ: `logiBUS::io::DI::logiBUS_IX`)

  - Parameter: `QI` = `TRUE`, `Input` = `Input_I1`
  - Ausgang: `IN` (digitaler Wert), `IND` (Ereignis bei Flanke)
- **Input_CD** (Typ: `logiBUS::io::DI::logiBUS_IX`)
  - Parameter: `QI` = `TRUE`, `Input` = `Input_I2`
- **Input_R** (Typ: `logiBUS::io::DI::logiBUS_IX`)
  - Parameter: `QI` = `TRUE`, `Input` = `Input_I3`
- **Input_LD** (Typ: `logiBUS::io::DI::logiBUS_IX`)
  - Parameter: `QI` = `TRUE`, `Input` = `Input_I4`
- **Output_QU** (Typ: `logiBUS::io::DQ::logiBUS_QX`)
  - Parameter: `QI` = `TRUE`, `Output` = `Output_Q1`
  - Eingang: `OUT` (Wert), `REQ` (Ereignis)
- **Output_QD** (Typ: `logiBUS::io::DQ::logiBUS_QX`)
  - Parameter: `QI` = `TRUE`, `Output` = `Output_Q2`
- **Q_NumericValue** (Typ: `isobus::UT::Q::Q_NumericValue`)
  - Parameter: `u16ObjId` = `OutputNumber_N1`
  - Eingang: `REQ` (Ereignis), `u32NewValue` (numerischer Wert)
  - **Funktionsweise**: Dieser Baustein gibt einen 32-Bit-Wert (unsigned) auf eine numerische Anzeige im Terminal aus. Die Objekt-ID verweist auf einen vordefinierten Ausgabekanal.
- **F_LINT_TO_UDINT** (Typ: `iec61131::conversion::F_LINT_TO_UDINT`)
  - Eingang: `REQ` (Ereignis), `IN` (Wert vom Typ `LINT`)
  - Ausgang: `CNF` (Ereignis), `OUT` (Wert vom Typ `UDINT`)
  - **Funktionsweise**: Wandelt einen 64-Bit-Ganzzahlwert (LINT) in einen 32-Bit-vorzeichenlosen Wert (UDINT) um. **Hinweis**: Negative Eingangswerte können nicht korrekt dargestellt werden, da der Zielbereich nur positive Zahlen zulässt. Die Übung enthält einen Kommentar, der auf diese Einschränkung hinweist.

Programmablauf und Verbindungen

Die Steuerung erfolgt über vier digitale Eingänge (logiBUS), die als Taster dienen:

- **Input_I1** (CU): Aufwärtszählen
- **Input_I2** (CD): Abwärtszählen
- **Input_I3** (R): Zähler zurücksetzen auf 0
- **Input_I4** (LD): Zähler auf Voreinstellwert PV (=10) laden

Jeder Eingangsbaustein erzeugt bei einer positiven Flanke ein Ereignis (`IND`). Diese Ereignisse werden auf den Ereigniseingang `REQ` des Zählerbausteins `FB_CTUD_LINT` geschaltet. Der Zähler reagiert dann auf den entsprechenden Dateneingang (`CU`, `CD`, `R`, `LD`).

Die Datenverbindungen übertragen die logischen Zustände der Taster:

- `Input_CU.IN` ⟶ `FB_CTUD_LINT.CU`
- `Input_CD.IN` ⟶ `FB_CTUD_LINT.CD`
- `Input_R.IN` ⟶ `FB_CTUD_LINT.R`
- `Input_LD.IN` ⟶ `FB_CTUD_LINT.LD`

Nach einer Verarbeitung des Zählers wird das Bestätigungsereignis (`CNF`) an mehrere Ausgänge weitergeleitet:

- `Output_QU.REQ` und `Output_QD.REQ` ⟶ aktualisiert die digitalen Ausgänge
- `F_LINT_TO_UDINT.REQ` ⟶ startet die Typumwandlung

Die Datenausgänge `QU` und `QD` steuern die digitalen Ausgänge:

- `FB_CTUD_LINT.QU` ⟶ `Output_QU.OUT` (Ausgangsklemme `Output_Q1`)
- `FB_CTUD_LINT.QD` ⟶ `Output_QD.OUT` (Ausgangsklemme `Output_Q2`)

Der aktuelle Zählerstand (`CV`) wird über die Umwandlung `F_LINT_TO_UDINT` geleitet:

- `FB_CTUD_LINT.CV` ⟶ `F_LINT_TO_UDINT.IN`

- `F_LINT_TO_UDINT.OUT` ⟶ `Q_NumericValue.u32NewValue`

Nach erfolgreicher Umwandlung löst `F_LINT_TO_UDINT.CNF` das Ereignis `Q_NumericValue.REQ` aus, wodurch der Wert auf dem Terminal angezeigt wird.

Lernziele:

- Einführung in IEC 61131-3 Zählerbausteine (CTUD) mit großer Datenbreite
- Verwendung von logiBUS-Ein-/Ausgabebaugruppen
- Typkonvertierung und deren Fallstricke (keine negativen Zahlen bei UDINT)
- Ereignisgesteuerte Ablaufsteuerung in 4diac

Hinweis: Die Übung setzt Grundkenntnisse der 4diac-IDE und der logiBUS-Konfiguration voraus. Die Eingänge müssen mit realen oder simulierten Tastern verbunden werden; die Ausgänge mit Leuchten oder Statusanzeigen.

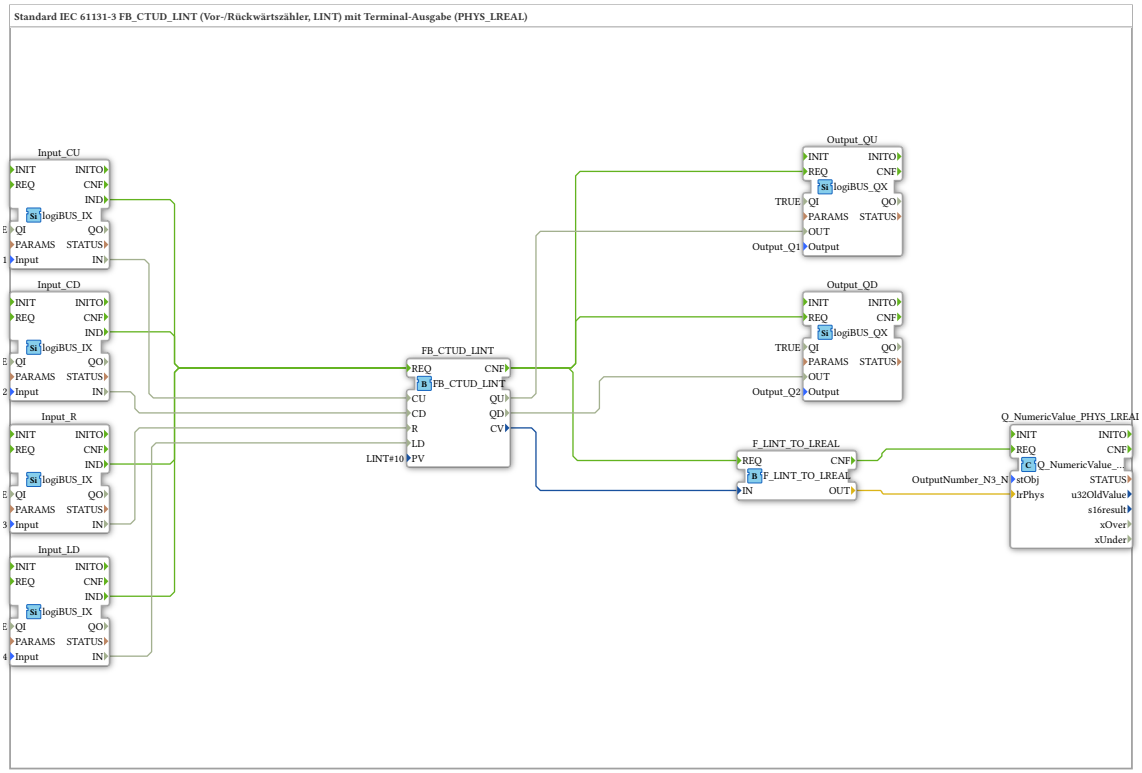
Zusammenfassung

Die Übung 222 realisiert einen universellen Vor-/Rückwärtszähler (FB_CTUD_LINT) mit einem 64-Bit-Zählbereich, gesteuert durch vier Tastereingänge. Zwei digitale Ausgänge zeigen die Zustände `QU` (Maximum erreicht) und `QD` (Minimum erreicht) an, während ein Terminal den aktuellen Zahlwert ausgibt. Die notwendige Typumwandlung von `LINT` nach `UDINT` ist bewusst als Problemfall dokumentiert, um auf die mögliche Fehlinterpretation negativer Werte hinzuweisen. Der Entwurf folgt dem IEC-61131-3-Standard und erlaubt eine einfache Erweiterung für andere Zählparameter.

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

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

2.2.1.400 Uebung_222b: Standard IEC 61131-3 FB_CTUD_LINT (Vor-/Rückwärtszähler, LINT) mit Terminal-Ausgabe (PHYS_LREAL)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung realisiert einen standardkonformen IEC 61131-3 Vor-/Rückwärtszähler mit dem Datentyp **LINT** (64-Bit Ganzzahl). Der Zähler wird über vier digitale Eingänge gesteuert: Hochzählen (CU), Herun-

terzählen (CD), Rücksetzen (R) und Laden des Presetwerts (LD). Der aktuelle Zählerstand wird an zwei digitale Ausgänge (QU und QD) sowie über einen Datentypkonverter als physikalischer Gleitkommawert (LREAL) auf einem Terminal ausgegeben. Damit kann der Zählerstand im laufenden Betrieb direkt in der Entwicklungsumgebung beobachtet werden.

Verwendete Funktionsbausteine (FBs)

Bausteinname	Typ	Parameter	Beschreibung
FB_CTUD_LINT	iec61131::counters::FB_CTUD_LINT	PV = LINT#10	Vor-/Rückwärtszähler (LINT). Zählt bei CU-Ereignissen hoch, bei CD-Ereignissen runter. Ein R-Ereignis setzt den Zähler auf 0, ein LD-Ereignis lädt den Presetwert PV.
Input_CU	logiBUS::io::DI::logiBUS_IOI	QI = TRUE, Input = Input_I1	Digitaler Eingang (logiBUS) – Signal zum Hochzählen.
Input_CD	logiBUS::io::DI::logiBUS_IOI	QI = TRUE, Input = Input_I2	Digitaler Eingang – Signal zum Herunterzählen.
Input_R	logiBUS::io::DI::logiBUS_IOI	QI = TRUE, Input = Input_I3	Digitaler Eingang – Rücksetzsignal.
Input_LD	logiBUS::io::DI::logiBUS_IOI	QI = TRUE, Input = Input_I4	Digitaler Eingang – Ladesignal für Presetwert.
Output_QU	logiBUS::io::DQ::logiBUS_QOI	QI = TRUE, Output = Output_Q1	Digitaler Ausgang – wird aktiv, wenn der Zählerstand ≥ PV ist.
Output_QD	logiBUS::io::DQ::logiBUS_QOI	QI = TRUE, Output = Output_Q2	Digitaler Ausgang – wird aktiv, wenn der Zählerstand ≤ 0 ist.
F_LINT_TO_LREAL	iec61131::conversion::F_LINT_TO_LREAL	–	Konvertiert den aktuellen Zählerstand (LINT) in den Gleitkommadatentyp LREAL.
Q_NumericValue_PHYS_LREAL	PHYS_LREAL::Q::Q_NumericValue_PHYS_LREAL	stObj = OutputNumber_N3	Gibt den konvertierten Wert als physikalische LREAL-Größe auf dem Terminal (OutputNumber_N3) aus.

Programmablauf und Verbindungen

1. **Ereignisverkettung**
   - Jeder Tastendruck an einem der vier Eingänge (Input_CU, Input_CD, Input_R, Input_LD) löst das Ereignis IND aus.
   - Diese Ereignisse werden alle an den Ereigniseingang REQ des Zählers FB_CTUD_LINT geleitet.
   - Nach der Abarbeitung (Ereignisausgang CNF) wird der Folgebaustein Output_QU und Output_QD sowie der Konverter F_LINT_TO_LREAL angestoßen.
   - Nach der Konvertierung wird das Terminal-Objekt Q_NumericValue_PHYS_LREAL aktualisiert.
2. **Datenverkettung**

- Die digitalen Eingangssignale (IN) werden direkt an die entsprechenden Zählereingänge gelegt:
  - Input_CU.IN → FB_CTUD_LINT.CU
  - Input_CD.IN → FB_CTUD_LINT.CD
  - Input_R.IN → FB_CTUD_LINT.R
  - Input_LD.IN → FB_CTUD_LINT.LD
- Die Zählerausgänge:
  - FB_CTUD_LINT.QU → Output_QU.OUT (Schaltet Ausgang Q1)
  - FB_CTUD_LINT.QD → Output_QD.OUT (Schaltet Ausgang Q2)
- Der aktuelle Zählerstand (CV, LINT) wird über den Konverter in LREAL umgewandelt:
  - FB_CTUD_LINT.CV → F_LINT_TO_LREAL.IN
  - F_LINT_TO_LREAL.OUT → Q_NumericValue_PHYS_LREAL.lrPhys
- Das Terminal zeigt somit den Zählerstand als Dezimalzahl mit Gleitkomma an.

3. **Lernziele**

- Verständnis des IEC 61131-3 CTUD-Funktionsbausteins (LINT).
- Umgang mit digitalen Ein-/Ausgängen im logiBUS-System.
- Datentypkonvertierung von LINT nach LREAL.
- Visualisierung von Prozesswerten über ein Terminal-Objekt.

4. **Schwierigkeitsgrad & Vorkenntnisse**

- **Schwierigkeit:** Mittel.
- **Vorkenntnisse:** Grundlagen der 4diac-IDE, Umgang mit SubApp-Typen, Ereignis- und Datenverbindungen, einfache IEC 61131-3 Kenntnisse.

5. **Ausführung**

- Laden Sie die Übung in die 4diac-IDE.
- Weisen Sie die entsprechenden logiBUS-Kanäle zu (Input_I1 … I4, Output_Q1, Q2).
- Starten Sie die Simulation oder übertragen Sie sie auf die Zielhardware.
- Beobachten Sie die Zählerstände auf dem Terminal (OutputNumber_N3) und die Ausgänge Q1, Q2.

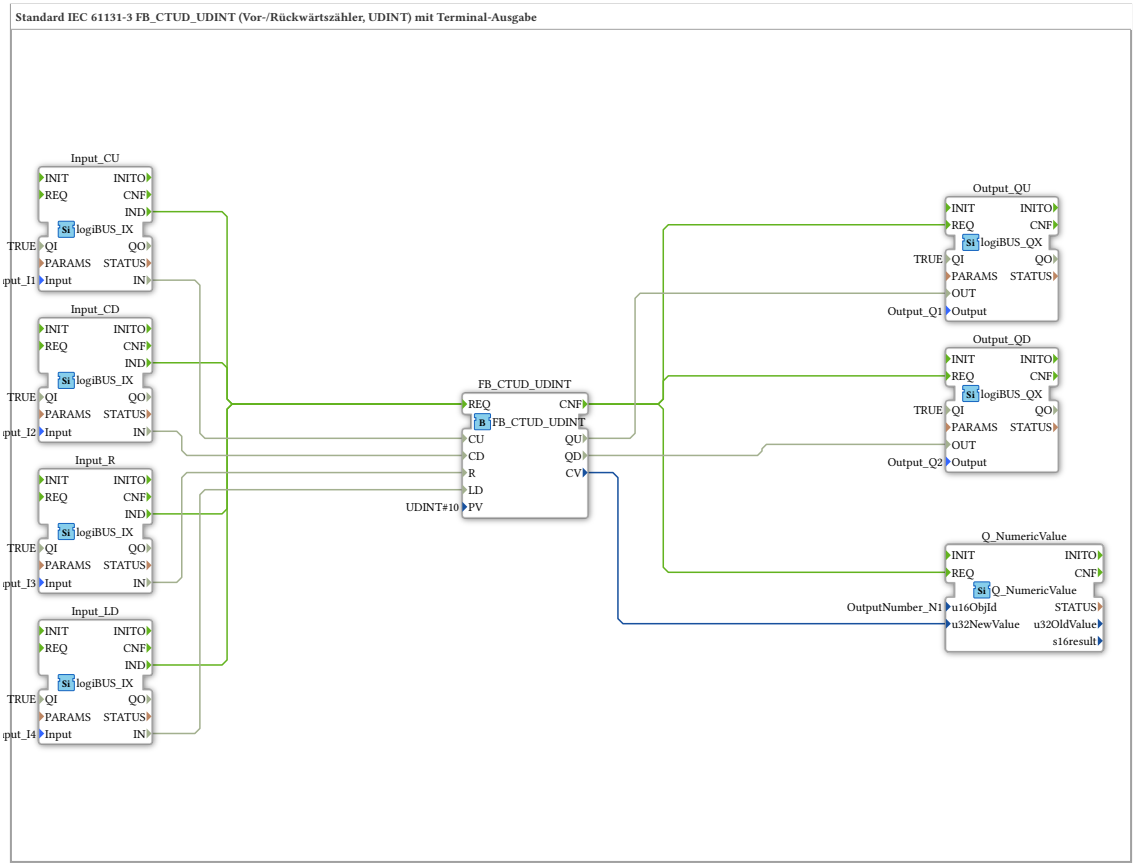
Zusammenfassung

Die Übung „Uebung_222b“ demonstriert einen vollständigen IEC 61131-3 konformen Vor-/Rückwärtszähler mit LINT-Datentyp. Durch die Kombination digitaler Ein-/Ausgänge, eines Standardzählers und einer Datentypkonvertierung wird ein einfacher, aber praxisnaher Zähleraufbau realisiert, dessen aktueller Wert über ein Terminal beobachtet werden kann. Dieses Beispiel eignet sich besonders zum Erlernen von Ereignisverknüpfungen und der Datenkonvertierung zwischen Ganzzahl- und Gleitkommatypen in 4diac.

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

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

2.2.1.401 Uebung_223: Standard IEC 61131-3 FB_CTUD_UDINT (Vor-/Rückwärtszähler, UDINT) mit Terminal-Ausgabe



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung implementiert einen Vor-/Rückwärtszähler (FB_CTUD_UDINT) gemäß IEC 61131-3 mit Datentyp UDINT. Der Zähler wird über digitale Eingänge gesteuert und gibt den aktuellen Zählerstand sowie zwei Ausgangssignale (QU, QD) an digitale Ausgänge aus. Zusätzlich wird der Zählerwert auf einem Terminal (Q_NumericValue) ausgegeben.

Die Übung demonstriert die Verwendung von logiBUS-Ein-/Ausgangsbausteinen in Verbindung mit einem standardisierten Zähler-FB. Ein Kommentar im Netzwerk weist darauf hin, dass ggf. E_D_FF-Bausteine eingefügt werden sollten, um die Ereignisanzahl zu reduzieren.

Verwendete Funktionsbausteine (FBs)

In der SubApp werden folgende Funktionsbausteine verwendet:

- **FB_CTUD_UDINT** – IEC 61131-3 Vor-/Rückwärtszähler (UDINT)
  - Parameter: `PV` = `UDINT#10`
  - Ereigniseingang: `REQ`
  - Dateneingänge: `CU` (Vorwärts zählen), `CD` (Rückwärts zählen), `R` (Rücksetzen), `LD` (Laden des Wertes von PV)
  - Datenausgänge: `QU` (Zählerstand >= PV), `QD` (Zählerstand = 0), `CV` (aktueller Zählerwert, UDINT)
- **Input_CU** – Digitaler Eingang (logiBUS_IX)
  - Parameter: `QI` = TRUE, `Input` = `Input_I1`
- **Input_CD** – Digitaler Eingang (logiBUS_IX)
  - Parameter: `QI` = TRUE, `Input` = `Input_I2`
- **Input_R** – Digitaler Eingang (logiBUS_IX)

  - Parameter: `QI` = TRUE, `Input` = `Input_I3`
- **Input_LD** – Digitaler Eingang (logiBUS_IX)
  - Parameter: `QI` = TRUE, `Input` = `Input_I4`
- **Output_QU** – Digitaler Ausgang (logiBUS_QX)
  - Parameter: `QI` = TRUE, `Output` = `Output_Q1`
- **Output_QD** – Digitaler Ausgang (logiBUS_QX)
  - Parameter: `QI` = TRUE, `Output` = `Output_Q2`
- **Q_NumericValue** – Terminalausgabe (isobus::UT::Q::Q_NumericValue)
  - Parameter: `u16ObjId` = `OutputNumber_N1`
  - Daten: `u32NewValue` (erhält den Zählerwert CV)

Programmablauf und Verbindungen

Die Übung ist als eine SubApp realisiert, die keine eigenen Ein-/Ausgangsschnittstellen besitzt, sondern direkt auf die globalen logiBUS- und ISOBUS-Ressourcen zugreift.

Ereignisverbindungen:

1. Alle vier digitalen Eingabebausteine (Input_CU, Input_CD, Input_R, Input_LD) lösen bei einer steigenden Flanke (IND-Ereignis) das Ereignis `REQ` des Zählers FB_CTUD_UDINT aus.
2. Nach Bearbeitung des Zählers wird dessen Bestätigungsereignis `CNF` an die Ausgabebausteine Output_QU, Output_QD und Q_NumericValue weitergeleitet, sodass diese ihre Werte aktualisieren.

Datenverbindungen:

- `Input_CU.IN` ⟶ `FB_CTUD_UDINT.CU`
- `Input_CD.IN` ⟶ `FB_CTUD_UDINT.CD`
- `Input_R.IN` ⟶ `FB_CTUD_UDINT.R`
- `Input_LD.IN` ⟶ `FB_CTUD_UDINT.LD`
- `FB_CTUD_UDINT.QU` ⟶ `Output_QU.OUT`
- `FB_CTUD_UDINT.QD` ⟶ `Output_QD.OUT`
- `FB_CTUD_UDINT.CV` ⟶ `Q_NumericValue.u32NewValue`

Ablauf:

- Sobald ein digitaler Eingang (z.B. Taster) betätigt wird, wird das entsprechende Signal an den Zähler weitergegeben.
- Der Zähler zählt bei jedem positiven Flanke an CU hoch, bei CD runter. Mit R wird der Zähler auf 0 zurückgesetzt. Mit LD wird der aktuelle Zählerwert auf den Wert von PV (10) gesetzt.
- Die Ausgänge QU und QD zeigen an, ob der Zählerstand den PV-Wert erreicht hat (QU) bzw. ob der Zählerstand 0 ist (QD).
- Der aktuelle Zählerstand (CV) wird zusätzlich auf einem Terminal ausgegeben (Q_NumericValue).

Hinweis: Ein Kommentar im Netzwerk schlägt vor, ggf. einen oder zwei E_D_FF-Bausteine einzufügen, um die Ereignisse zu reduzieren, wenn mehrere Eingänge gleichzeitig ausgelöst werden.

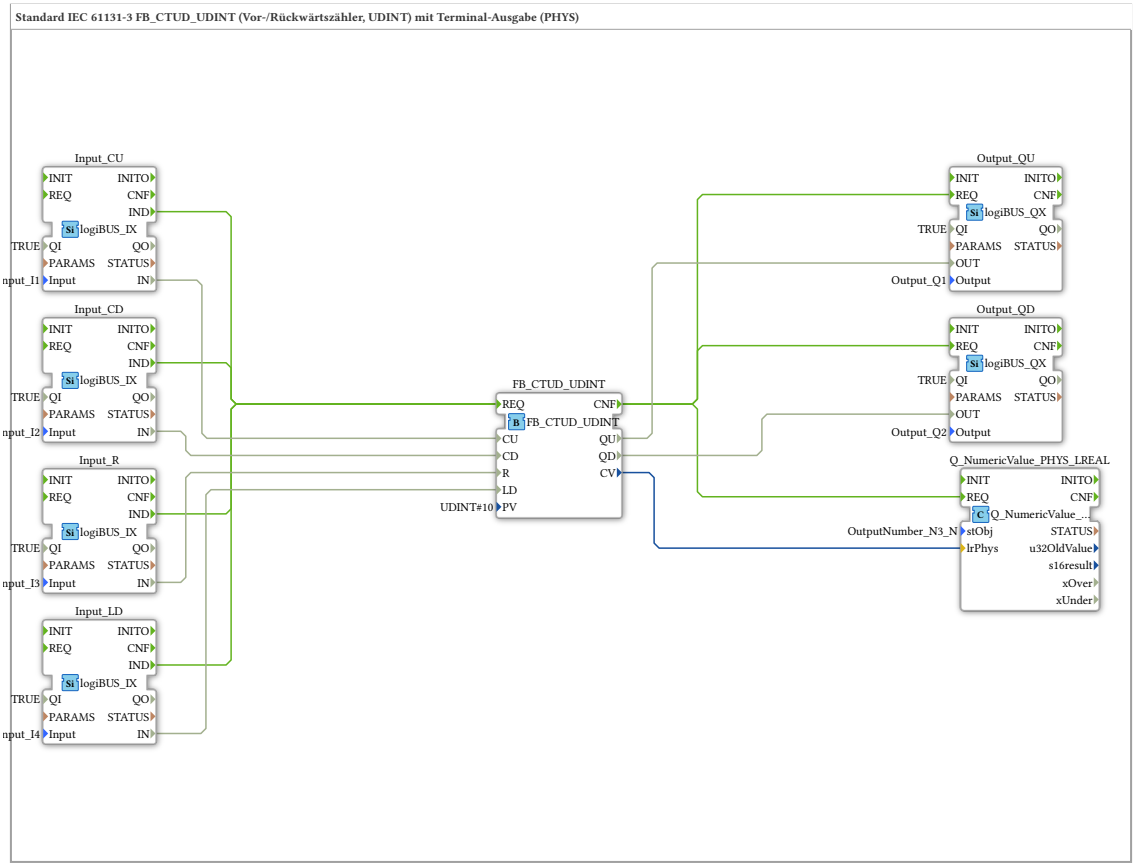
Zusammenfassung

Die Übung 223 zeigt die praktische Anwendung eines IEC 61131-3 Vor-/Rückwärtszählers (FB_CTUD_UDINT) in einer 4diac-IDE-Umgebung. Durch die Kombination von logiBUS-Eingängen und -Ausgängen sowie einer Terminalausgabe wird ein vollständiger Zähler mit Anzeige realisiert. Die Übung verdeutlicht die Verbindung von Ereignis- und Datenflüssen zwischen Funktionsbausteinen unterschiedlicher Bibliotheken.

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

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

2.2.1.402 Uebung_223b: Standard IEC 61131-3 FB_CTUD_UDINT (Vor-/Rückwärtszähler, UDINT) mit Terminal-Ausgabe (PHYS)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung implementiert einen vorwärts/rückwärts zählenden Funktionsbaustein nach IEC 61131-3 (Typ `FB_CTUD_UDINT`) mit einem Wertebereich vom Typ `UDINT`. Der aktuelle Zählerstand wird zusätzlich auf einem Terminal (PHYS) ausgegeben. Die Steuerung der Zählerfunktionen erfolgt über vier digitale Eingänge, und zwei digitale Ausgänge signalisieren die Zähler-Richtungsanzeigen.

Verwendete Funktionsbausteine (FBs)

- **FB_CTUD_UDINT**
  - Typ: `iec61131::counters::FB_CTUD_UDINT`
  - Parameter: `PV` = `UDINT#10`
  - Funktion: Standard IEC 61131-3 Vor-/Rückwärtszähler (UDINT). Zählt bei jedem positiven Flanke an `CU` hoch, an `CD` runter. Über `R` wird der Zähler auf 0 zurückgesetzt, über `LD` wird der Wert aus `PV` geladen.
- **Input_CU, Input_CD, Input_R, Input_LD**
  - Typ: `logiBUS::io::DI::logiBUS_IX`
  - Parameter:
    - `QI` = `TRUE`
    - `Input` = `Input_I1` (für CU), `Input_I2` (für CD), `Input_I3` (für R), `Input_I4` (für LD)
  - Funktion: Digitaleingangsklemmen zur Erfassung der Taster- oder Sensorsignale.
- **Output_QU, Output_QD**
  - Typ: `logiBUS::io::DQ::logiBUS_QX`
  - Parameter:
    - `QI` = `TRUE`

    - `Output` = `Output_Q1` (für QU), `Output_Q2` (für QD)
  - Funktion: Digitalausgangsklemmen zur Anzeige, ob der Zähler den Wert `PV` erreicht hat (QU) bzw. ob der Wert 0 erreicht hat (QD).
- **Q_NumericValue_PHYS_LREAL**
  - Typ: `isobus::UT::Q::Q_NumericValue_PHYS_LREAL`
  - Parameter: `stObj` = `OutputNumber_N3`
  - Funktion: Gibt den aktuellen Zählerwert als `LREAL` auf dem Terminal (PHYS) aus. Der Wert wird direkt von `CV` (UDINT) übernommen – eine Typkonvertierung ist aufgrund der automatischen Umwandlung von UDINT nach LREAL nicht erforderlich.

Programmablauf und Verbindungen

Die Ereignissteuerung erfolgt über die `IND`-Ereignisse der digitalen Eingänge. Jeder Tastendruck an einem Eingang löst eine `REQ`-Verarbeitung des Zählers aus:

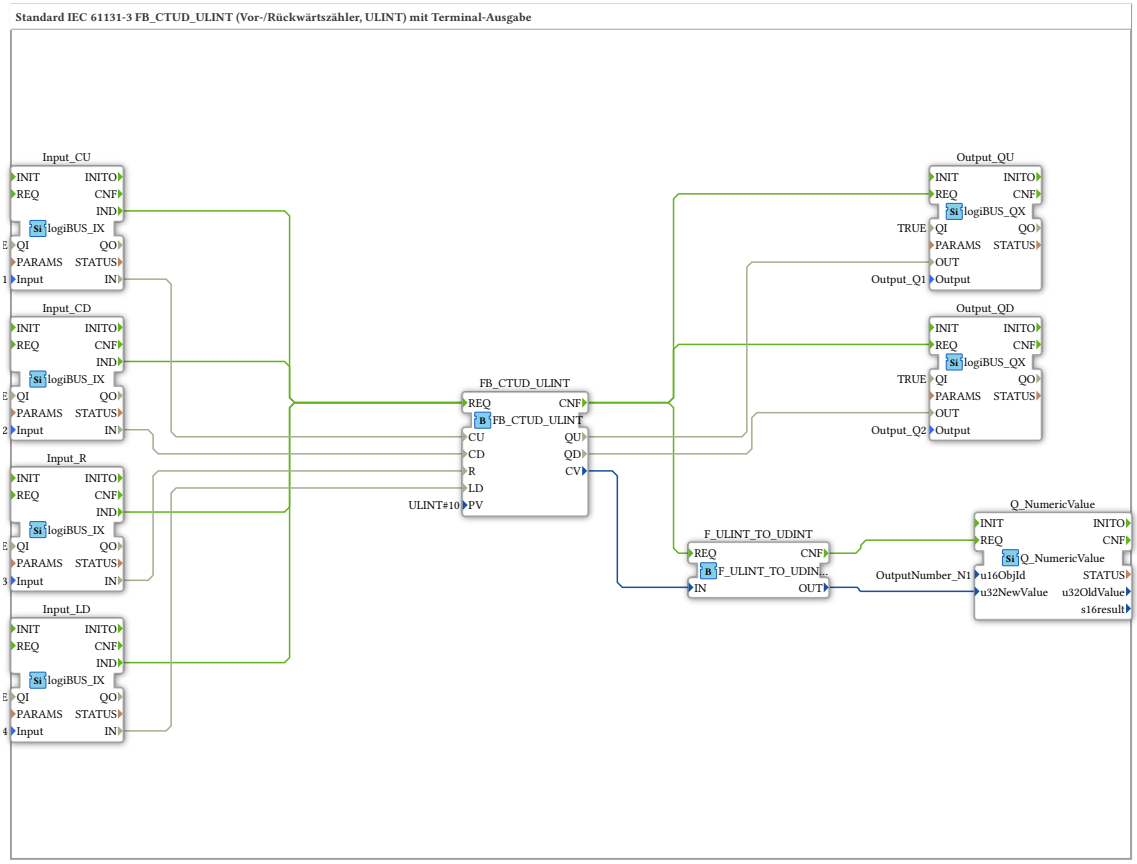
1. **Ereignisverbindungen**
   - `Input_CU.IND` ⟶ `FB_CTUD_UDINT.REQ`
   - `Input_CD.IND` ⟶ `FB_CTUD_UDINT.REQ`
   - `Input_R.IND` ⟶ `FB_CTUD_UDINT.REQ`
   - `Input_LD.IND` ⟶ `FB_CTUD_UDINT.REQ`
   - `FB_CTUD_UDINT.CNF` ⟶ `Output_QU.REQ`, `Output_QD.REQ`, `Q_NumericValue_PHYS_LREAL.REQ`
2. **Datenverbindungen**
   - `Input_CU.IN` ⟶ `FB_CTUD_UDINT.CU`
   - `Input_CD.IN` ⟶ `FB_CTUD_UDINT.CD`
   - `Input_R.IN` ⟶ `FB_CTUD_UDINT.R`
   - `Input_LD.IN` ⟶ `FB_CTUD_UDINT.LD`
   - `FB_CTUD_UDINT.QU` ⟶ `Output_QU.OUT`
   - `FB_CTUD_UDINT.QD` ⟶ `Output_QD.OUT`
   - `FB_CTUD_UDINT.CV` ⟶ `Q_NumericValue_PHYS_LREAL.lrPhys`

Somit wird bei jeder Zustandsänderung eines Eingangs der Zähler neu berechnet, die Ausgänge aktualisiert und der aktuelle numerische Wert auf dem Terminal ausgegeben. Der Kommentar im Netzwerk weist darauf hin, dass `UDINT` ohne explizite Konvertierung an den `LREAL`-Eingang angeschlossen werden kann.

Zusammenfassung

Die Übung 223b demonstriert den Einsatz eines standardisierten IEC 61131-3 Vor-/Rückwärtszählers (`FB_CTUD_UDINT`) in einer Steuerungsumgebung. Vier digitale Eingänge ermöglichen das Hochzählen, Herunterzählen, Rücksetzen und Laden des Zählerwerts. Zwei digitale Ausgänge zeigen das Erreichen des Grenzwerts (`QU`) bzw. des Nullwerts (`QD`) an. Zusätzlich wird der aktuelle Zählerstand als physikalischer Wert auf einem Terminal (PHYS) ausgegeben. Die einfache Verkabelung und die direkte Typumwandlung von `UDINT` auf `LREAL` machen die Schaltung besonders übersichtlich.

2.2.1.403 Uebung_224: Standard IEC 61131-3 FB_CTUD_ULINT (Vor-/Rückwärtszähler, ULINT) mit Terminal-Ausgabe



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung implementiert einen kombinierten Vor-/Rückwärtszähler nach IEC 61131-3 (Typ `FB_CTUD_ULINT`) mit 64-Bit-Vorwahl (ULINT). Der Zählerstand wird über den Terminal-Baustein `Q_NumericValue` auf einer numerischen Anzeige ausgegeben. Die Eingänge werden über logiBUS-Digitaleingänge bereitgestellt, die Ausgänge über logiBUS-Digitalausgänge.

Verwendete Funktionsbausteine (FBs)

- **FB_CTUD_ULINT** – Typ: `iec61131::counters::FB_CTUD_ULINT`
  - Parameter: `PV` = `ULINT#10` (Vorwahlwert)
  - Eingänge: `CU` (Vorwärts zählen), `CD` (Rückwärts zählen), `R` (Reset), `LD` (Laden von PV)
  - Ausgänge: `QU` (Vorwahl erreicht, vorwärts), `QD` (Vorwahl erreicht, rückwärts), `CV` (aktueller Zählerwert)
  - Ereignisse: `REQ` (Aufruf), `CNF` (Bestätigung)
  - Funktionsweise: Der Baustein zählt bei jedem positiven Flanke an `CU` (vorwärts) bzw. `CD` (rückwärts). Bei `R=TRUE` wird der Zähler auf 0 gesetzt, bei `LD=TRUE` wird der Wert von `PV` geladen. Bei Erreichen des Vorwahlwerts werden die Ausgänge `QU`/`QD` gesetzt.
- **Input_CU** – Typ: `logiBUS::io::DI::logiBUS_IX`
  - Parameter: `QI = TRUE`, `Input = Input_I1`
  - Eingänge: `IND` (Ereignis bei Änderung), `IN` (Datenwert)
  - Funktionsweise: digitaler Eingang, der den physischen Eingang `Input_I1` auf den Zähleingang `CU` legt.
- **Input_CD** – Typ: `logiBUS::io::DI::logiBUS_IX`
  - Parameter: `QI = TRUE`, `Input = Input_I2`

  - Funktionsweise: digitaler Eingang für den Rückwärtszähleingang `CD`.
- **Input_R** – Typ: `logiBUS::io::DI::logiBUS_IX`
  - Parameter: `QI = TRUE`, `Input = Input_I3`
  - Funktionsweise: digitaler Eingang für den Reseteingang `R`.
- **Input_LD** – Typ: `logiBUS::io::DI::logiBUS_IX`
  - Parameter: `QI = TRUE`, `Input = Input_I4`
  - Funktionsweise: digitaler Eingang für den Ladeeingang `LD`.
- **Output_QU** – Typ: `logiBUS::io::DQ::logiBUS_QX`
  - Parameter: `QI = TRUE`, `Output = Output_Q1`
  - Funktionsweise: digitaler Ausgang, der den Zustand von `QU` an den physischen Ausgang `Output_Q1` weitergibt.
- **Output_QD** – Typ: `logiBUS::io::DQ::logiBUS_QX`
  - Parameter: `QI = TRUE`, `Output = Output_Q2`
  - Funktionsweise: digitaler Ausgang, der den Zustand von `QD` an den physischen Ausgang `Output_Q2` weitergibt.
- **F_ULINT_TO_UDINT** – Typ: `iec61131::conversion::F_ULINT_TO_UDINT`
  - Eingänge: `IN` (ULINT), Ausgänge: `OUT` (UDINT)
  - Ereignisse: `REQ`, `CNF`
  - Funktionsweise: konvertiert den 64-Bit-Zählerwert (`CV`) in einen 32-Bit-Wert. **Achtung:** bei Werten $> 2^{32}$ kann es zum Überlauf kommen (siehe Kommentar).
- **Q_NumericValue** – Typ: `isobus::UT::Q::Q_NumericValue`
  - Parameter: `u16ObjId = OutputNumber_N1`
  - Eingänge: `REQ`, `u32NewValue`
  - Funktionsweise: gibt den übergebenen Zahlenwert auf dem Terminal (numerische Anzeige) aus.

Programmablauf und Verbindungen

1. **Ereignissteuerung** Jeder digitale Eingang (Input_CU…Input_LD) erzeugt bei Zustandsänderung ein Ereignis (`IND`). Alle diese Ereignisse sind auf den `REQ`-Eingang des Zählers `FB_CTUD_ULINT` geschaltet. Dadurch wird der Zähler bei jedem Tastendruck an einem der vier Eingänge aufgerufen. *Hinweis:* Da gleichzeitige Ereignisse mehrerer Eingänge zu einem einzigen Aufruf zusammengefasst werden (oder auch nicht), kann es zu unerwünschtem Verhalten kommen. Im Kommentar wird daher empfohlen, ggf. einen oder zwei `E_D_FF` (Event-DFlipFlop) einzubauen, um die Ereignisse zu reduzieren.
2. **Datenverbindungen**
   - Die digitalen Eingangswerte (`IN`) werden direkt auf die entsprechenden Zählereingänge (`CU`, `CD`, `R`, `LD`) geführt.
   - Die Ausgänge `QU` und `QD` des Zählers sind mit den Digitalausgängen (`Output_QU`, `Output_QD`) verbunden.
   - Der aktuelle Zählerwert `CV` wird über den Konvertierungsbaustein `F_ULINT_TO_UDINT` auf 32 Bit reduziert und an den Terminal-Baustein `Q_NumericValue` übergeben.
   - Die Ereigniskette: `FB_CTUD_ULINT.CNF` löst gleichzeitig die Ausgangs-FBs und die Konvertierung aus. Nach der Konvertierung wird `Q_NumericValue` aktualisiert.
3. **Parameter** Der Vorwahlwert `PV` ist auf `ULINT#10` gesetzt – ein Vergleich mit diesem Wert setzt die Ausgänge `QU`/`QD`.
4. **Lernziele**
   - Kennenlernen des IEC-61131-3-Zählers `FB_CTUD_ULINT`.
   - Umgang mit digitalen Ein-/Ausgängen über logiBUS.
   - Verwendung von Konvertierungsbausteinen und Terminalausgabe.
   - Bewusstsein für Ereigniskollisionen und mögliche Abhilfe durch `E_D_FF`.
5. **Schwierigkeitsgrad:** Mittel **Voraussetzungen:** Grundlegende Kenntnisse der 4diac-IDE, Umgang mit logiBUS-E/A-Bausteinen und Ereignisverkabelung.

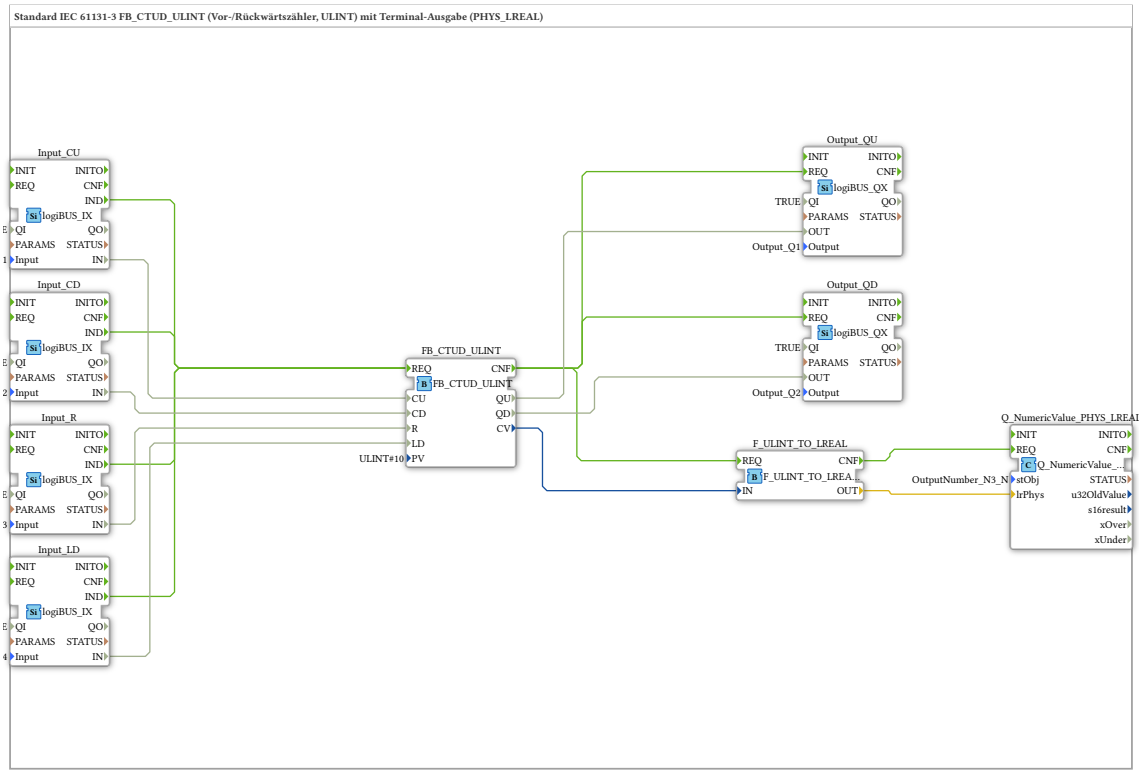
Zusammenfassung

Die Übung 224 realisiert einen vollständigen Vor-/Rückwärtszähler mit 64-Bit-Auflösung und Terminalanzeige. Vier digitale Eingänge steuern Zählen, Reset und Laden eines Vorwahlwerts. Die Ausgänge signalisieren das Erreichen der Vorwahl. Die optionalen `E_D_FF` dienen der Stabilisierung bei mehreren gleichzeitigen Ereignissen. Dieses Beispiel vermittelt den praktischen Einsatz von Zählern, E/A-Anbindung und Datentypkonvertierung in 4diac.

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

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

2.2.1.404 Uebung_224b: Standard IEC 61131-3 FB_CTUD_ULINT (Vor-/Rückwärtszähler, ULINT) mit Terminal-Ausgabe (PHYS_LREAL)



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung implementiert einen Vor-/Rückwärtszähler nach IEC 61131-3 (Funktionsbaustein `FB_CTUD_ULINT`) mit dem Datentyp `ULINT`. Der aktuelle Zählerstand wird über einen Konverter in einen physikalischen Wert (`LREAL`) umgewandelt und auf einer Terminalausgabe (z. B. einem Bedienpanel) angezeigt. Die Steuerung erfolgt über vier digitale Eingänge (CU, CD, R, LD), zwei digitale Ausgänge zeigen die Grenzwertsignale (QU, QD) an.

Verwendete Funktionsbausteine (FBs)

- **`FB_CTUD_ULINT`** (Typ: `iec61131::counters::FB_CTUD_ULINT`)
  - Parameter: `PV` = `ULINT#10`
  - Ereigniseingang: `REQ` (trigger)
  - Ereignisausgang: `CNF`
  - Dateneingänge: `CU` (count up), `CD` (count down), `R` (reset), `LD` (load)
  - Datenausgänge: `QU` (Grenzwert oben), `QD` (Grenzwert unten), `CV` (aktueller Zählerstand)
  - Funktionsweise: Realisiert einen Vor-/Rückwärtszähler. Bei jedem ansteigenden Ereignis auf `CU` wird der Zähler erhöht, bei `CD` verringert. Bei `R` wird der Zähler auf 0 gesetzt, bei `LD` wird der Wert von

PV geladen. Die Ausgänge QU und QD werden gesetzt, wenn der Zählerstand den programmierten Grenzwert erreicht hat.

- **Input_CU** (Typ: logiBUS::io::DI::logiBUS_IX)
  - Parameter: QI = TRUE, Input = Input_I1
  - Funktionsweise: Digitaler Eingang, der den physischen Eingang I1 des logiBUS-Moduls einliest.
- **Input_CD** (Typ: logiBUS::io::DI::logiBUS_IX)
  - Parameter: QI = TRUE, Input = Input_I2
  - Funktionsweise: Digitaler Eingang für I2.
- **Input_R** (Typ: logiBUS::io::DI::logiBUS_IX)
  - Parameter: QI = TRUE, Input = Input_I3
  - Funktionsweise: Digitaler Eingang für I3.
- **Input_LD** (Typ: logiBUS::io::DI::logiBUS_IX)
  - Parameter: QI = TRUE, Input = Input_I4
  - Funktionsweise: Digitaler Eingang für I4.
- **Output_QU** (Typ: logiBUS::io::DQ::logiBUS_QX)
  - Parameter: QI = TRUE, Output = Output_Q1
  - Funktionsweise: Digitaler Ausgang, der den physischen Ausgang Q1 ansteuert.
- **Output_QD** (Typ: logiBUS::io::DQ::logiBUS_QX)
  - Parameter: QI = TRUE, Output = Output_Q2
  - Funktionsweise: Digitaler Ausgang für Q2.
- **F_ULINT_TO_LREAL** (Typ: iec61131::conversion::F_ULINT_TO_LREAL)
  - Funktionsweise: Konvertiert den ULINT-Wert des Zählerstands in einen LREAL-Wert für die Ausgabe.
- **Q_NumericValue_PHYS_LREAL** (Typ: isobus::UT::Q::Q_NumericValue_PHYS_LREAL)
  - Parameter: stObj = OutputNumber_N3 (Verweis auf ein Terminalobjekt)
  - Funktionsweise: Stellt einen numerischen Wert als physikalische Größe (LREAL) auf dem Terminal dar.

Programmablauf und Verbindungen

Der Ablauf wird ereignisgesteuert über die **Ereignisverbindungen** gesteuert:

- Jeder digitale Eingang (Input_CU, Input_CD, Input_R, Input_LD) erzeugt ein IND-Ereignis, wenn sich der Eingangszustand ändert.
- Diese vier Ereignisse werden gemeinsam auf den REQ-Ereigniseingang des Zählers FB_CTUD_ULINT geführt.
- Nach der Verarbeitung des Zählers (bei jedem Ereignis) erzeugt dieser ein CNF-Ereignis. Dieses Ereignis wird parallel an drei Bausteine weitergegeben:
  - Output_QU.REQ – aktualisiert den digitalen Ausgang QU
  - Output_QD.REQ – aktualisiert den digitalen Ausgang QD
  - F_ULINT_TO_LREAL.REQ – startet die Konvertierung des Zählerstands
- Nach Abschluss der Konvertierung erzeugt F_ULINT_TO_LREAL ein CNF-Ereignis, das an Q_NumericValue_PHYS_LREAL.REQ weitergeleitet wird, um die Anzeige zu aktualisieren.

Die **Datenverbindungen** übertragen die Werte:

- Die digitalen Eingangssignale (Input_*.IN) werden direkt auf die entsprechenden Dateneingänge des Zählers geführt: CU, CD, R, LD.
- Die Ausgangssignale des Zählers (QU, QD) werden an die digitalen Ausgänge Output_QU.OUT und Output_QD.OUT angeschlossen.
- Der aktuelle Zählerstand (CV) wird in den Konverter F_ULINT_TO_LREAL.IN eingespeist.
- Das konvertierte LREAL-Signal (OUT) gelangt zum Terminalbaustein Q_NumericValue_PHYS_LREAL.lrPhys und wird dort angezeigt.

Hinweis: Im Netzwerk befindet sich ein Kommentar, der darauf hinweist, dass bei Bedarf ein oder zwei E_D_FF-Bausteine eingefügt werden können, um die Ereignishäufigkeit zu reduzieren. Dies kann sinnvoll sein, wenn mehrere Eingänge gleichzeitig schalten.

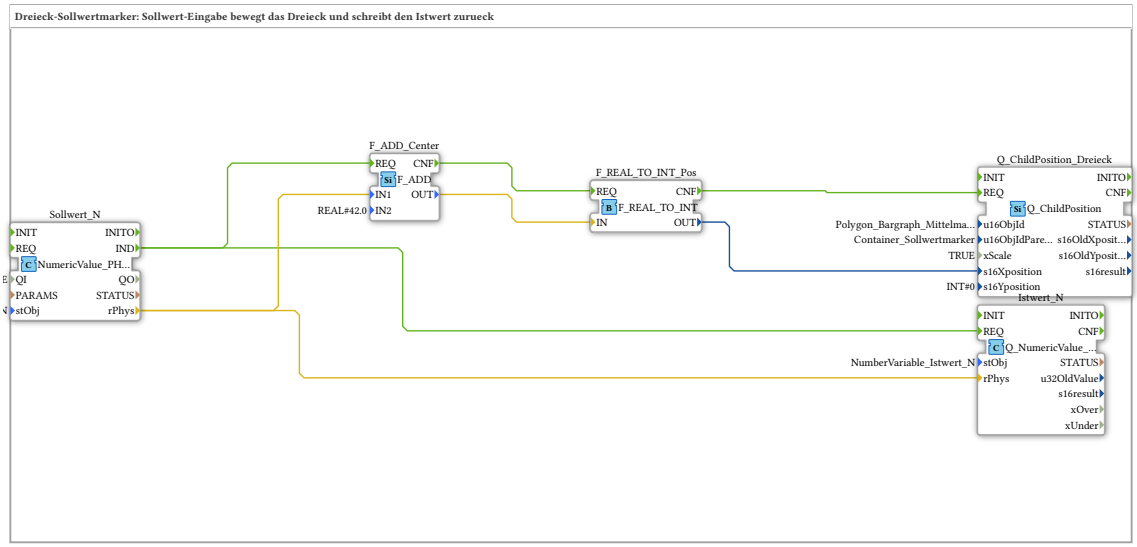
Zusammenfassung

Die Übung demonstriert die Verwendung eines IEC 61131-3 Vor-/Rückwärtszählers (`FB_CTUD_ULINT`) in 4diac. Die digitale Hardware (Eingänge I1–I4, Ausgänge Q1–Q2) wird über logiBUS-Bausteine angebunden. Der Zählerstand wird in einen physikalischen Messwert (`LREAL`) konvertiert und auf einem Terminal ausgegeben. Die Lernziele umfassen den Umgang mit Zählerbausteinen, Ereignisverkettung, Typkonvertierung und die Anbindung von Ein-/Ausgängen in einer ereignisgesteuerten Umgebung.

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

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

2.2.1.405 Uebung_225: Dreieck-Sollwertmarker



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung positioniert ein kleines Dreieck (Mittelmarker) innerhalb eines Containers auf dem Virtual Terminal, abhängig von einem über ein Eingabefeld eingegebenen Sollwert (ISO 11783-6 Annex F.16, Change Child Position). Zusätzlich wird derselbe Sollwert unverändert als Istwert zurückgeschrieben, sodass Eingabefeld und Bargraph-Zeiger synchron denselben Wert anzeigen.

Verwendete Funktionsbausteine (FBs)

- **Sollwert_N**: Terminal-Eingabe (Typ: `isobus::UT::io::NumericValue::NumericValue_PHYS`)
  - **Parameter**: `QI = TRUE`, `stObj = NumberVariable_Sollwert_N`
  - **Erklärung**: Liest Änderungen des über `InputNumber_Sollwert` (VT-Objekt 9000) eingegebenen Werts. Das VT meldet Wertänderungen eines an eine `NumberVariable` gebundenen Eingabefelds unter der Objekt-ID der Variable (`NumberVariable_Sollwert`, 21000), nicht unter der ID des Eingabefelds selbst. Liefert den physikalischen Wert (bereits um −42 versetzt, Bereich −42…+42) als `REAL` über `rPhys`.
- **F_ADD_Center**: Addition (Typ: `iec61131::arithmetic::F_ADD`)
  - **Parameter**: `IN2 = REAL#42.0`
  - **Erklärung**: Addiert den Center-Offset von 42 auf den Sollwert, um ihn auf die tatsächliche Pixel-X-Position (0…84) innerhalb des Containers umzurechnen.
- **F_REAL_TO_INT_Pos**: Typkonvertierung (Typ: `iec61131::conversion::F_REAL_TO_INT`)
  - **Parameter**: keine
  - **Erklärung**: Wandelt den berechneten `REAL`-Positionswert in einen `INT`-Pixelwert um.
- **Q_ChildPosition_Dreieck**: Objektpositionierung (Typ: `isobus::UT::Q::Q_ChildPosition`)
  - **Parameter**: `u16ObjId = Polygon_Bargraph_Mittelmarker` (Kind-Objekt, Dreieck), `u16ObjIdParent = Container_Sollwertmarker` (Parent-Objekt), `xScale = TRUE`, `s16Yposition = 0`

  - **Erklärung**: Schreibt die neue X-Position des Dreiecks innerhalb des Containers per Change-Child-Position-Kommando. `xScale = TRUE` sorgt dafür, dass der Pixel-Offset zusätzlich mit dem DataMask-Skalierungsfaktor multipliziert wird. Y bleibt konstant 0.
- **Istwert_N**: Terminal-Ausgabe (Typ: `isobus::UT::Q::Q_NumericValue_PHYS`)
  - **Parameter**: `stObj = NumberVariable_Istwert_N`
  - **Erklärung**: Schreibt denselben physikalischen Sollwert unverändert als Istwert zurück; dadurch aktualisieren sich sowohl `InputNumber_Istwert` als auch der bestehende Bargraph-Positionszeiger.

Sub-Bausteine: keine

Programmablauf und Verbindungen

1. Ändert der Bediener `InputNumber_Sollwert`, feuert `Sollwert_N.IND` und liefert den physikalischen Wert über `Sollwert_N.rPhys`.
2. Das Ereignis `Sollwert_N.IND` löst zwei parallele Ketten aus: einerseits `F_ADD_Center.REQ` (Positionsberechnung), andererseits direkt `Istwert_N.REQ` (Istwert-Rückschreibung).
3. `Sollwert_N.rPhys` wird auf `F_ADD_Center.IN1` geführt; `F_ADD_Center` addiert `REAL#42.0` und gibt das Ergebnis über `OUT` weiter, sobald `F_ADD_Center.CNF` `F_REAL_TO_INT_Pos.REQ` auslöst.
4. `F_REAL_TO_INT_Pos` wandelt den `REAL`-Wert in `INT` um; sein `CNF`-Ereignis löst `Q_ChildPosition_Dreieck.REQ` aus, während der Datenwert über `OUT` auf `s16Xposition` geführt wird.
5. `Q_ChildPosition_Dreieck` schreibt die neue Dreieck-Position auf das VT.
6. Parallel dazu wird `Sollwert_N.rPhys` unverändert auf `Istwert_N.rPhys` geführt und bei `Istwert_N.REQ` als Istwert zurückgeschrieben.

Zusammenfassung

Die Übung demonstriert die klassische, Schritt-für-Schritt verdrahtete Umsetzung einer Virtual-Terminal-Objektpositionierung: Sollwert lesen, auf Pixelkoordinaten umrechnen (Offset-Addition + Typkonvertierung), Position schreiben und parallel den Istwert zurückmelden. Für die vollständig über Adapter verdrahtete Variante siehe `Uebung_225_AX` in `test_AX`, die dieselbe Funktion mit einer reinen AR/AI-Adapterkette statt einzelner Event-/DataConnections umsetzt.

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

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

2.2.1.406 Übung 225: Dreieck-Sollwertmarker

Thema: Virtual-Terminal-Objektpositionierung (ISO 11783-6 Annex F.16)

Situationsbeschreibung

Im ISO-Designer-Pool `Workspace_Dreieck` zeigt ein kleines Dreieck (`Polygon_Bargraph_Mittelmarker`) innerhalb eines 96×14 px großen Containers (`Container_Sollwertmarker`) eine Sollposition an — analog zum Mittelmarker, wie er in vergleichbaren Anwendungen für Achslenkung/Hangausgleich verwendet wird. Der Bediener gibt über das Eingabefeld `InputNumber_Sollwert` (VT-Objekt 9000) einen Wert zwischen **−42 und +42** ein (0 = Mitte, ISO-Designer wendet den Offset −42 bereits objektseitig an). Das Dreieck soll dieser Eingabe sofort folgen, und der eingestellte Wert soll zusätzlich als Istwert (`NumberVariable_Istwert`, treibt sowohl `InputNumber_Istwert` als auch den Bargraph-Zeiger) zurückgemeldet werden.

Funktionsbeschreibung

- **Sollwert lesen:** Ändert der Bediener `InputNumber_Sollwert`, liefert das VT den physikalischen Wert (bereits skaliert/versetzt gemäß Offset −42) als `REAL`.
- **Dreieck bewegen:** Der Sollwert (-42…+42) muss auf die tatsächliche Pixel-X-Position (0…84) innerhalb des Containers umgerechnet werden (+42, dann `REAL`→`INT`) und per **Change Child Position** (ISO 11783-6 Annex F.16) auf das Polygon-Objekt geschrieben werden — Kind-Objekt ist das Dreieck (`Polygon_Bargraph_Mittelmarker`, ID 16000, gültig als Kind: TypPolygon 16000–16999), Parent-Objekt ist der Container (`Container_Sollwertmarker`, ID 3000, gültig als Parent: Container, Annex B.4). Y bleibt konstant 0.
- **Istwert zurückschreiben:** Derselbe physikalische Sollwert wird unverändert (gleicher Offset, keine weitere Umrechnung nötig) auf `NumberVariable_Istwert` geschrieben — dadurch aktualisieren sich sowohl `InputNumber_Istwert` als auch der Bargraph-Positionszeiger gleichzeitig.

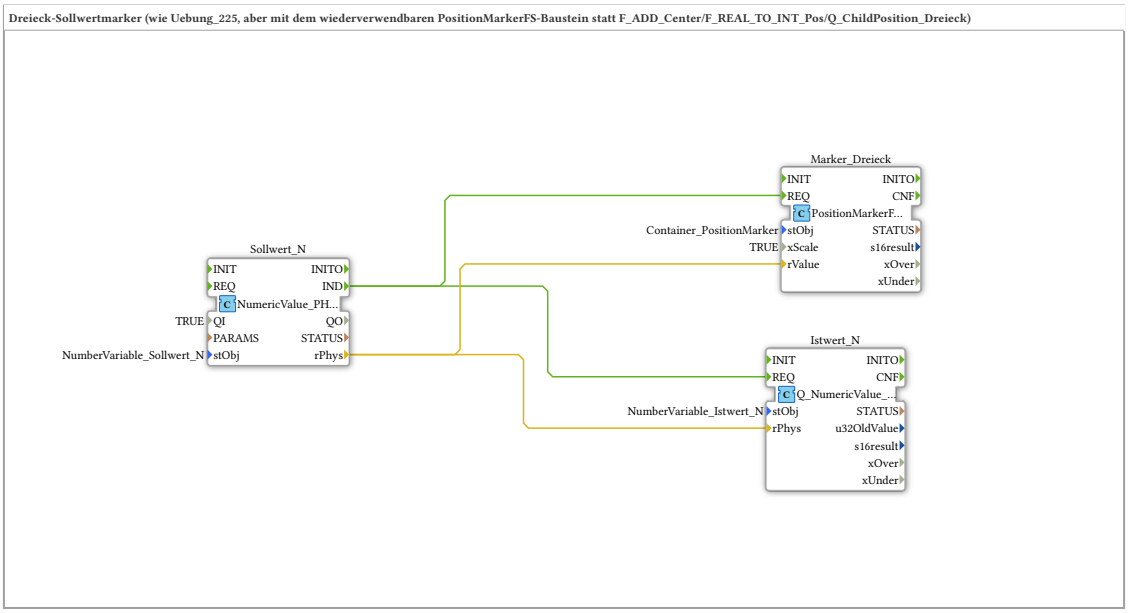
Arbeitsauftrag

1. Legen Sie die SubApp `Uebung_225` an (bereits als Referenzlösung vorhanden).
2. Lesen Sie `InputNumber_Sollwert` (VT-Objekt 9000) mit einer `NumericValue_PHYS`-Instanz (`stObj := NumberVariable_Sollwert_N` aus `DefaultPool_Dreieck_Numeric.gcf`) — das VT meldet Wertänderungen eines an eine `NumberVariable` gebundenen Eingabefelds unter der Objekt-ID der Variable (hier `NumberVariable_Sollwert`, 21000), nicht unter der ID des Eingabefelds selbst (9000).
3. Rechnen Sie den physikalischen Wert per `F_ADD` (+42.0) und `F_REAL_TO_INT` auf die Pixel-X-Position (0…84) um.
4. Schreiben Sie die neue Position mit `Q_ChildPosition` (`u16ObjId := Polygon_Bargraph_Mittelmarker`, `u16ObjIdParent := Container_Sollwertmarker`, `s16Yposition := 0`).
5. Schreiben Sie parallel denselben physikalischen Sollwert mit `Q_NumericValue_PHYS` (`stObj := NumberVariable_Istwert_N`) als Istwert zurück.
6. Am echten Terminal testen: Sollwert ändern → Dreieck folgt sofort, Istwert-Feld und Bargraph-Zeiger zeigen denselben Wert.

Referenzlösung

`Uebung_225.SUB` — validiert gegen `fbtype.xsd`/`subapp.xsd` (siehe `iec61499-creator`-Skill). Für die vollständig über Adapter verdrahtete Variante siehe `Uebung_225_AX` in `test_AX`.

2.2.1.407 Uebung_225b: Dreieck-Sollwertmarker mit PositionMarkerFS



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung ist funktional identisch zu `Uebung_225`, ersetzt aber die dort einzeln verdrahteten Bausteine `F_ADD_Center`/`F_REAL_TO_INT_Pos`/`Q_ChildPosition_Dreieck` durch den neuen, wiederverwendbaren Baustein `PositionMarkerFS` (`isobus::UT::Q`). `Uebung_225` selbst bleibt dabei unverändert als eigenständige Referenz bestehen.

Verwendete Funktionsbausteine (FBs)

- **Sollwert_N**: Terminal-Eingabe (Typ: `isobus::UT::io::NumericValue::NumericValue_PHYS`)
  - **Parameter**: `QI = TRUE`, `stObj = NumberVariable_Sollwert_N`
  - **Erklärung**: Liest Änderungen des über `InputNumber_Sollwert` (VT-Objekt 9000) eingegebenen Werts und liefert ihn als physikalischen `REAL`-Wert über `rPhys` (Bereich −42…+42).
- **Marker_Dreieck**: Positionsmarker (Typ: `isobus::UT::Q::PositionMarkerFS`)
  - **Parameter**: `stObj = Container_PositionMarker`, `xScale = TRUE`
  - **Erklärung**: Übernimmt die komplette Bewegung des Dreiecks: addiert intern den Center-Offset, klammert das Ergebnis auf den gültigen Bewegungsbereich (mit `xOver`/`xUnder`-Rückmeldung), rech-

net `REAL` ⟶ `INT` um und schreibt die Position per Change Child Position auf das Polygon-Objekt. Kind-ID, Parent-ID, Bewegungsbereich und Center-Offset kommen gebündelt aus der Konstante `Container_PositionMarker` (Typ `PositionMarker_S`), die automatisch aus der `.jop`-Geometrie erzeugt wird. `xScale = TRUE` aktiviert die Multiplikation des Pixel-Offsets mit dem DataMask-Skalierungsfaktor.

- **Istwert_N**: Terminal-Ausgabe (Typ: `isobus::UT::Q::Q_NumericValue_PHYS`)
  - **Parameter**: `stObj = NumberVariable_Istwert_N`
  - **Erklärung**: Schreibt denselben physikalischen Sollwert unverändert als Istwert zurück; aktualisiert `InputNumber_Istwert` und den bestehenden Bargraph-Positionszeiger.

Sub-Bausteine: keine

Der Baustein `PositionMarkerFS` selbst ist kein Composite-SubApp in dieser Übung, sondern eine vorgefertigte Bibliothekskomponente (`Ventilsteuerung\4diacIDE-workspace\.lib\isobus-3.0.0\typelib\UT\Q\PositionMarkerFS.fbt`).

Programmablauf und Verbindungen

1. Ändert der Bediener `InputNumber_Sollwert`, feuert `Sollwert_N.IND` und liefert den physikalischen Wert über `Sollwert_N.rPhys`.
2. Das Ereignis `Sollwert_N.IND` triggert parallel `Marker_Dreieck.REQ` und `Istwert_N.REQ`.
3. `Sollwert_N.rPhys` wird direkt auf `Marker_Dreieck.rValue` geführt; `PositionMarkerFS` berechnet intern Offset, Klammerung und Pixelposition und schreibt sie auf das Dreieck-Objekt.
4. Parallel dazu wird `Sollwert_N.rPhys` unverändert auf `Istwert_N.rPhys` geführt und dort als Istwert zurückgeschrieben.

Zusammenfassung

Die Übung zeigt, wie sich eine wiederkehrende Aufgabe (Sollwertmarker positionieren, inklusive Klammerung und Bereichsüberwachung) in einem einzigen wiederverwendbaren Baustein (`PositionMarkerFS`) kapseln lässt, anstatt sie – wie in `Uebung_225` – jedes Mal aus Einzelbausteinen neu zusammenzusetzen. Für die vollständig über AR-Adapter verdrahtete Variante siehe `Uebung_225b_AX` in `test_AX`.

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

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

2.2.1.408 Übung 225b: Dreieck-Sollwertmarker mit PositionMarkerFS

Thema: Virtual-Terminal-Objektpositionierung (ISO 11783-6 Annex F.16) — wiederverwendbarer Baustein

Variante von Übung 225: dieselbe Aufgabe, aber statt die drei Bausteine `F_ADD_Center`/`F_REAL_TO_INT_Pos`/`Q_ChildPosition_Dreieck` einzeln zu verdrahten, wird der neue wiederverwendbare Baustein `PositionMarkerFS` (`isobus::UT::Q`) eingesetzt. Übung 225 selbst bleibt unverändert als eigenständige Referenz stehen.

Situationsbeschreibung

Im ISO-Designer-Pool `Workspace_Dreieck` zeigt ein kleines Dreieck (`Polygon_Bargraph_Mittelmarker`) innerhalb eines 96×14 px großen Containers (`Container_Sollwertmarker`) eine Sollposition an — analog zum Mittelmarker, wie er in vergleichbaren Anwendungen für Achslenkung/Hangausgleich verwendet wird. Der Bediener gibt über das Eingabefeld `InputNumber_Sollwert` (VT-Objekt 9000) einen Wert zwischen **−42 und +42** ein (0 = Mitte, ISO-Designer wendet den Offset −42 bereits objektseitig an). Das Dreieck soll dieser Eingabe sofort folgen, und der eingestellte Wert soll zusätzlich als Istwert (`NumberVariable_Istwert`, treibt sowohl `InputNumber_Istwert` als auch den Bargraph-Zeiger) zurückgemeldet werden.

Funktionsbeschreibung

- **Sollwert lesen:** Ändert der Bediener `InputNumber_Sollwert`, liefert das VT den physikalischen Wert (bereits skaliert/versetzt gemäß Offset −42) als `REAL`.
- **Dreieck bewegen:** übernimmt der wiederverwendbare Baustein `PositionMarkerFS` (`isobus::UT::Q`) — er addiert intern den Center-Offset, klammert das Ergebnis auf den gültigen Bewegungsbereich (mit `xOver`/`xUnder`-Rückmeldung), rechnet `REAL`⟶`INT` um und schreibt die Position per **Change**

Child Position (ISO 11783-6 Annex F.16) auf das Polygon-Objekt. Kind-Objekt ist das Dreieck (`Polygon_Bargraph_Mittelmarker`, ID 16000, gültig als Kind: TypPolygon 16000–16999), Parent-Objekt ist der Container (`Container_Sollwertmarker`, ID 3000, gültig als Parent: Container, Annex B.4). Y bleibt konstant 0. Kind-ID, Parent-ID, Bewegungsbereich und Center-Offset kommen gebündelt aus **einer** Konstante (`Container_PositionMarker`, Typ `PositionMarker_S`), die `GcfScript.py` automatisch aus der `.jop`-Geometrie erzeugt (Container ObjectName endet auf `_Sollwertmarker`) — analog zu `ScrollObjectPool_S` beim Scroll-Baustein.

- **Istwert zurückschreiben:** Derselbe physikalische Sollwert wird unverändert (gleicher Offset, keine weitere Umrechnung nötig) auf `NumberVariable_Istwert` geschrieben — dadurch aktualisieren sich sowohl `InputNumber_Istwert` als auch der Bargraph-Positionszeiger gleichzeitig.
- **Wichtig:** der Sollwert-Eingang von `PositionMarkerFS` (`rValue`) ist ein generischer `REAL`-Wert — er muss nicht zwingend von `NumericValue_PHYS` kommen, sondern kann z. B. auch direkt von einem Sensor gespeist werden. Deshalb bleibt `NumericValue_PHYS` bewusst außerhalb des Bausteins.

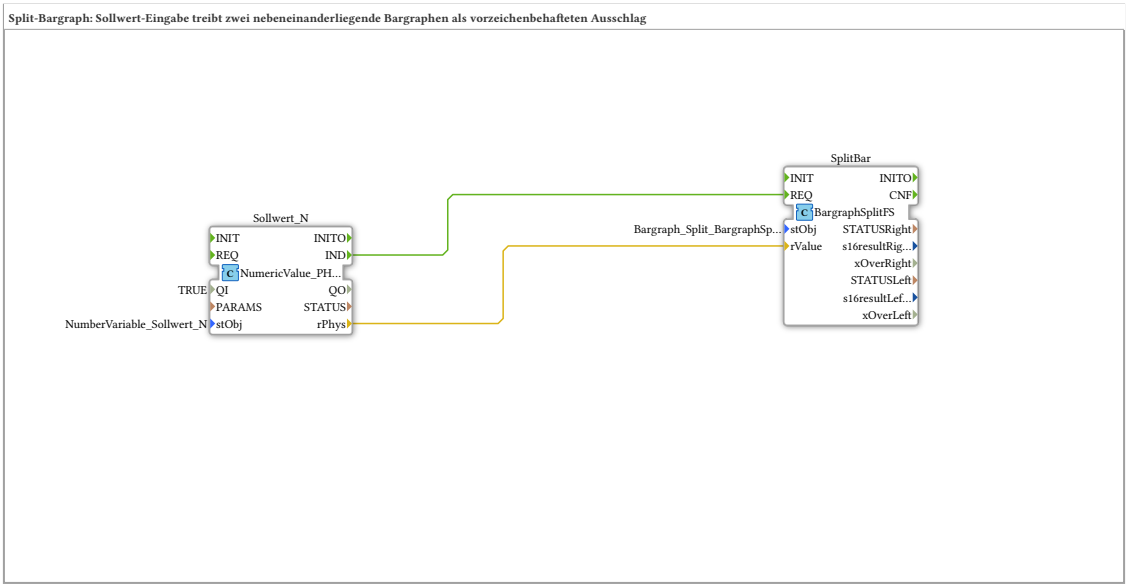
Arbeitsauftrag

1. Legen Sie die SubApp `Uebung_225b` an (bereits als Referenzlösung vorhanden).
2. Lesen Sie `InputNumber_Sollwert` (VT-Objekt 9000) mit einer `NumericValue_PHYS`-Instanz (`stObj := NumberVariable_Sollwert_N` aus `DefaultPool_Dreieck_Numeric.gcf`) — das VT meldet Wertänderungen eines an eine `NumberVariable` gebundenen Eingabefelds unter der Objekt-ID der Variable (hier `NumberVariable_Sollwert`, 21000), nicht unter der ID des Eingabefelds selbst (9000).
3. Bewegen Sie das Dreieck mit einer `PositionMarkerFS`-Instanz (`stObj := Container_PositionMarker` aus `DefaultPool_Dreieck_PositionMarker.gcf`, `xScale := TRUE`, `rValue :=` der physikalische Sollwert aus Schritt 2).
4. Schreiben Sie parallel denselben physikalischen Sollwert mit `Q_NumericValue_PHYS` (`stObj := NumberVariable_Istwert_N`) als Istwert zurück.
5. Am echten Terminal testen: Sollwert ändern → Dreieck folgt sofort, Istwert-Feld und Bargraph-Zeiger zeigen denselben Wert, `xOver`/`xUnder` bleiben `FALSE` solange der Sollwert im gültigen Bereich liegt.

Referenzlösung

`Uebung_225b.SUB` — validiert gegen `subapptype.xsd` (siehe `iec61499-creator`-Skill). Für die vollständig über Adapter verdrahtete Variante siehe `Uebung_225b_AX` in `test_AX`. Der wiederverwendbare Baustein liegt in `Ventilsteuerung\4diacIDE-workspace\.lib\isobus-3.0.0\typelib\UT\Q\PositionMarkerFS.fbt` (Struct `isobus::utils::childposition::PositionMarker_S`, Klammer-Hilfsbaustein `isobus::UT::Q::helpers::F_ClampReal`).

2.2.1.409 Uebung_226: Split-Bargraph



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung steuert zwei nebeneinanderliegende Linear-Bargraphen (`Bargraph_Split_links`/`Bargraph_Split_rechts`) so an, dass sie zusammen einen vorzeichenbehafteten Ausschlag von −42 bis +42 darstellen: bei positivem Sollwert füllt sich der rechte Balken, bei negativem der linke, der jeweils andere bleibt bei 0.

Verwendete Funktionsbausteine (FBs)

- **Sollwert_N**: Terminal-Eingabe (Typ: `isobus::UT::io::NumericValue::NumericValue_PHYS`)
  - **Parameter**: `QI = TRUE`, `stObj = NumberVariable_Sollwert_N`
  - **Erklärung**: Liest Änderungen des über `InputNumber_Sollwert` (VT-Objekt 9000) eingegebenen Werts und liefert ihn als physikalischen `REAL`-Wert über `rPhys`.
- **SplitBar**: Split-Bargraph-Ansteuerung (Typ: `isobus::UT::Q::BargraphSplitFS`)
  - **Parameter**: `stObj = Bargraph_Split_BargraphSplit`
  - **Erklärung**: Klammert den positiven Anteil des Sollwerts auf `[0, r32MaxMagnitude]` und schreibt ihn auf den rechten Balken; negiert parallel den Wert, klammert ihn ebenso und schreibt ihn auf den linken Balken. Beide Seiten schreiben direkt per Command Numeric Value (ISO 11783-6 Annex F.22) auf ihre Bargraph-Objekt-ID, ohne gebundene `NumberVariable`. Balken-IDs und der gemeinsame Magnitudenbereich kommen gebündelt aus der Konstante `Bargraph_Split_BargraphSplit` (Typ `BargraphSplit_S`), die automatisch aus der `.jop`-Geometrie erzeugt wird. `xOverRight`/`xOverLeft` melden nur eine Bereichsüberschreitung, wenn `|Sollwert| > r32MaxMagnitude` – es gibt bewusst keine `xUnder`-Ausgänge, da die jeweils inaktive Seite im Normalbetrieb ständig bei 0 liegt.

Sub-Bausteine: keine

Programmablauf und Verbindungen

1. Ändert der Bediener `InputNumber_Sollwert`, feuert `Sollwert_N.IND` und liefert den physikalischen Wert über `Sollwert_N.rPhys`.
2. `Sollwert_N.IND` triggert `SplitBar.REQ`.
3. `Sollwert_N.rPhys` wird direkt auf `SplitBar.rValue` geführt (kein Split-Baustein nötig, da nur ein einziger Verbraucher den Sollwert benötigt).
4. `SplitBar` berechnet daraus die beiden Balkenwerte (positiver Anteil rechts, negierter positiver Anteil links) und schreibt sie auf die beiden Bargraph-Objekte.

Zusammenfassung

Die Übung zeigt, wie ein einzelner vorzeichenbehafteter Sollwert über den wiederverwendbaren Baustein `BargraphSplitFS` auf zwei nebeneinanderliegende Bargraphen aufgeteilt wird, ohne dass die Klammerungs- und Vorzeichenlogik jedes Mal von Hand nachgebaut werden muss. Für die vollständig über AR-Adapter verdrahtete Variante siehe `Uebung_226_AX` in `test_AX`.

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

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

2.2.1.410 Übung 226: Split-Bargraph

Thema: Virtual-Terminal-Bargraphen (ISO 11783-6 Annex B.11.3) — vorzeichenbehafteter Ausschlag über zwei Balken

Situationsbeschreibung

Im ISO-Designer-Pool `Workspace_Dreieck` liegen zwei Linear-Bargraphen (`Bargraph_Split_links`, `Bargraph_Split_rechts`, VT-Objekte 18001/18002) unmittelbar nebeneinander — eine Darstellungsvariante, wie sie in echten Produktions-Pools für Signalgrößen mit Nulllage in der Mitte verwendet wird (dort mit anderen, kundenspezifischen Objektnamen; siehe `iso-designer-jop`-Skill zur allgemeinen Bargraph-Konfiguration). Jeder Balken zeigt `Min=0`/`Max=42` an. Zusammen bilden sie einen gemeinsamen, vorzeichenbehafteten Balken von **−42 bis +42**: bei positivem Wert füllt sich der rechte Balken von der Mitte nach außen, bei negativem Wert der linke, der jeweils andere bleibt bei 0. Der Bediener gibt über `InputNumber_Sollwert` (VT-Objekt 9000) einen Wert zwischen −42 und +42 ein.

Funktionsbeschreibung

- **Sollwert lesen:** Ändert der Bediener `InputNumber_Sollwert`, liefert das VT den physikalischen Wert als `REAL`.
- **Split-Bargraph ansteuern:** übernimmt der wiederverwendbare Baustein `BargraphSplitFS` (`isobus::UT::Q`) — er klammert den positiven Anteil des Werts auf `[0, r32MaxMagnitude]` und schreibt ihn auf den rechten Balken; parallel negiert er den Wert (`F_MUL` mit `-1.0`), klammert das Ergebnis genauso und schreibt es auf den linken Balken. Ist der Sollwert positiv, bleibt der linke Balken bei 0 (geklammert), und umgekehrt — das ist der normale Ruhezustand der jeweils inaktiven Seite, keine Fehlerbedingung. Beide Seiten schreiben direkt auf ihre Bargraph-Objekt-ID per **Command Numeric Value** (ISO 11783-6 Annex F.22, wie `Q_NumericValue_PHYS` es bereits für einzelne Zahlenfelder tut) — ohne gebundene `NumberVariable`. Balken-IDs und der gemeinsame Magnitudenbereich kommen gebündelt aus **einer** Konstante (`Bargraph_Split_BargraphSplit`, Typ `BargraphSplit_S`), die `GcfScript.py` automatisch aus der `.jop`-Geometrie erzeugt (zwei Bargraph-Objektnamen mit gemeinsamem Präfix, endend auf `_links`/`_rechts`).
- **Wichtig:** `xOverRight`/`xOverLeft` melden nur, wenn der Betrag des Sollwerts `r32MaxMagnitude` (42) tatsächlich überschreitet — es gibt bewusst **keine** `xUnder`-Ausgänge, da „Betrag unterhalb 0“ auf der gerade inaktiven Seite im Normalbetrieb ständig zutrifft und damit keine Diagnoseinformation wäre (anders als bei `PositionMarkerFS`, wo Rand-Werte selten sind).

Arbeitsauftrag

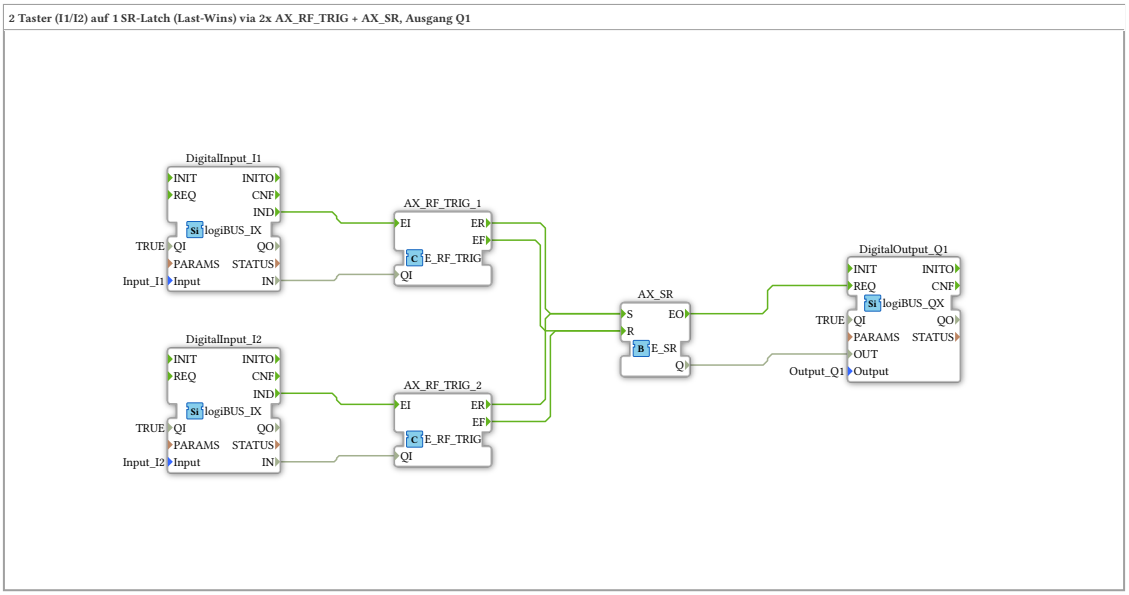
1. Legen Sie die SubApp `Uebung_226` an (bereits als Referenzlösung vorhanden).
2. Lesen Sie `InputNumber_Sollwert` (VT-Objekt 9000) mit einer `NumericValue_PHYS`-Instanz (`stObj := NumberVariable_Sollwert_N` aus `DefaultPool_Dreieck_Numeric.gcf`) — das VT meldet Wertänderungen eines an eine `NumberVariable` gebundenen Eingabefelds unter der Objekt-ID der Variable (hier `NumberVariable_Sollwert`, 21000), nicht unter der ID des Eingabefelds selbst (9000).
3. Steuern Sie den Split-Bargraph mit einer `BargraphSplitFS`-Instanz (`stObj := Bargraph_Split_BargraphSplit` aus `DefaultPool_Dreieck_BargraphSplit.gcf`, `rValue :=` der physikalische Sollwert aus Schritt 2).
4. Am echten Terminal testen: Sollwert positiv → rechter Balken füllt sich, linker bleibt leer; Sollwert negativ → umgekehrt; `xOverRight`/`xOverLeft` bleiben `FALSE` solange `|Sollwert| <= 42`.

Referenzlösung

`Uebung_226.SUB` — validiert gegen `subapptype.xsd` (siehe `iec61499-creator`-Skill). Für die vollständig über Adapter verdrahtete Variante siehe `Uebung_226_AX` in `test_AX`. Der wiederverwendbare Baustein liegt in `Ventilsteuerung\4diacIDE-workspace\.lib\isobus-3.0.0\typelib\UT\Q\BargraphSplitFS.fbt` (Struct `isobus::utils::bargraph::BargraphSplit_S`, Klammer-Hilfsbaustein `logiBUS::signalprocessing::clamp::F_ClampReal`).

2.2.1.411 Uebung_229: 2 Taster (I1/I2) auf 1 SR-Latch (Last-Wins) via 2x AX_RF_TRIG + AX_SR, Ausgang Q1

Dieser Artikel beschreibt die 4diac IDE Subapplikation Uebung_229 (2 Taster (I1/I2) auf 1 SR-Latch (Last-Wins) via 2x AX_RF_TRIG + AX_SR, Ausgang Q1).



Ziel der Übung

Das Ziel dieser Übung ist die Umsetzung folgender Anforderung: **2 Taster (I1/I2) auf 1 SR-Latch (Last-Wins) via 2x AX_RF_TRIG + AX_SR, Ausgang Q1**

Beschreibung und Komponenten

Die Übung besteht aus der Subapplikation Uebung_229.SUB, welche die folgende Bausteinstruktur verwendet:

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_I1**: Instanz des Typs logiBUS::io::DI::logiBUS_IX.
  - Parameter QI = TRUE
  - Parameter Input = Input_I1
- **DigitalInput_I2**: Instanz des Typs logiBUS::io::DI::logiBUS_IX.
  - Parameter QI = TRUE
  - Parameter Input = Input_I2
- **AX_RF_TRIG_1**: Instanz des Typs iec61499::events::E_RF_TRIG.
- **AX_RF_TRIG_2**: Instanz des Typs iec61499::events::E_RF_TRIG.
- **AX_SR**: Instanz des Typs iec61499::events::E_SR.
- **DigitalOutput_Q1**: Instanz des Typs logiBUS::io::DQ::logiBUS_QX.
  - Parameter QI = TRUE
  - Parameter Output = Output_Q1

Verbindungen und Schnittstellen

Ereignisverbindungen:

- AX_RF_TRIG_1.ER -> AX_SR.S
- AX_RF_TRIG_2.ER -> AX_SR.S
- AX_RF_TRIG_1.EF -> AX_SR.R
- AX_RF_TRIG_2.EF -> AX_SR.R
- DigitalInput_I2.IND -> AX_RF_TRIG_2.EI
- DigitalInput_I1.IND -> AX_RF_TRIG_1.EI
- AX_SR.EO -> DigitalOutput_Q1.REQ

Datenverbindungen:

- DigitalInput_I2.IN -> AX_RF_TRIG_2.QI
- DigitalInput_I1.IN -> AX_RF_TRIG_1.QI
- AX_SR.Q -> DigitalOutput_Q1.OUT

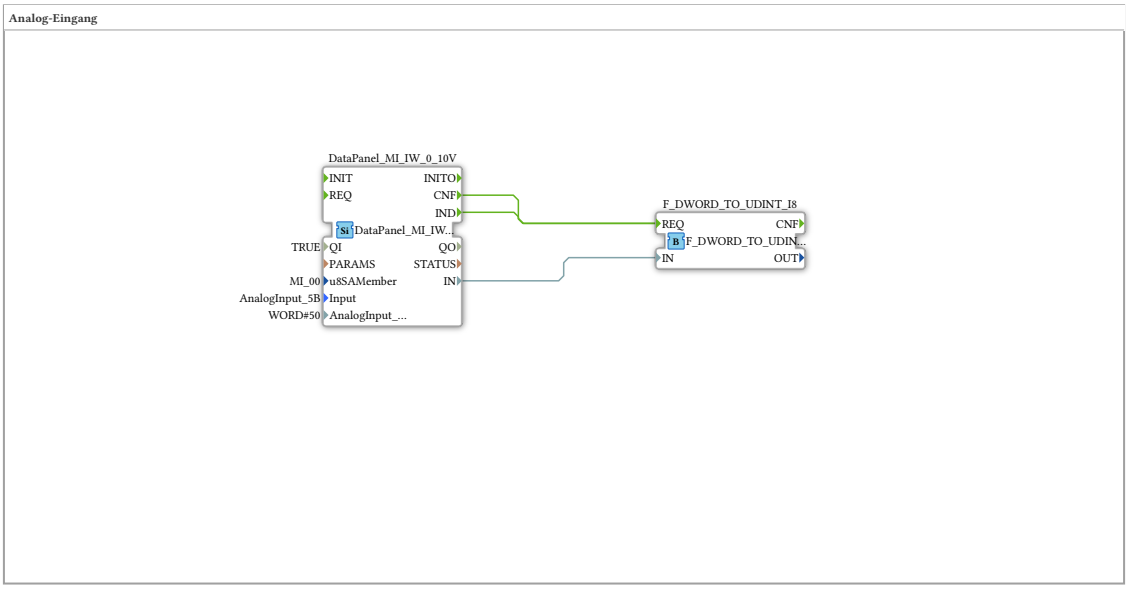
Programmablauf und Funktionsweise

1. **Initialisierung**: Beim Systemstart werden alle beteiligten Bausteine initialisiert.
2. **Ereignis- und Signalverarbeitung**: Zustandsänderungen an den Eingängen erzeugen Ereignisse, die über die definierten Verbindungen an die nachgelagerten Bausteine weitergeleitet werden.
3. **Ausgangsaktualisierung**: Die Zielbausteine verarbeiten die empfangenen Signale und aktualisieren entsprechend die Ausgänge.

Zusammenfassung

Die Übung Uebung_229 demonstriert anschaulich die modulare IEC 61499 Architektur in 4diac IDE.

2.2.1.412 Uebung_230: Analog-Eingang



• – ○ • – ○ • – ○ •

Einleitung

Diese Übung liest einen analogen 0-10V-Eingang auf dem DataPanel ein und wandelt den rohen DWORD-Messwert in einen nutzbaren UDINT-Zahlenwert um, gefiltert über eine Hysterese gegen Signalrauschen.

Verwendete Funktionsbausteine (FBs)

- **DataPanel_MI_IW_0_10V**: Analoger DataPanel-Eingang (Typ: `DataPanel::io::MI::AI::DataPanel_MI_IW_0_10V`)
  - **Parameter**: `QI = TRUE`, `u8SAMember = MI_00`, `Input = AnalogInput_5B`, `AnalogInput_hysteresis = 50`
  - **Erklärung**: Liest laufend die analoge Eingangsspannung (0-10V) auf dem DataPanel-Erweiterungsmodul ein, gefiltert über eine Hysterese von 50, um kleine Signalschwankungen zu unterdrücken.
- **F_DWORD_TO_UDINT_I8**: Typkonvertierung (Typ: `iec61131::conversion::F_DWORD_TO_UDINT`)
  - **Parameter**: keine
  - **Erklärung**: Wandelt den rohen DWORD-Messwert in einen UDINT-Zahlenwert um, der für die Weiterverarbeitung nutzbar ist.

Sub-Bausteine: keine

Programmablauf und Verbindungen

1. `DataPanel_MI_IW_0_10V` liest laufend die analoge Eingangsspannung ein.
2. Bei jeder Wertänderung (`IND`) wird `F_DWORD_TO_UDINT_I8.REQ` ausgelöst. `CNF` signalisiert danach den Abschluss der Konvertierung.
3. Der Datenwert `DataPanel_MI_IW_0_10V.IN` wird auf `F_DWORD_TO_UDINT_I8.IN` geführt.
4. `F_DWORD_TO_UDINT_I8` wandelt den DWORD-Rohwert in einen UDINT-Zahlenwert um.

Hinweis zur Hardware: Ein Kommentar im Netzwerk warnt ausdrücklich: Nicht jedes X-Board besitzt zwangsläufig die Analogeingänge `5A-8B/1A-4B` – manche Boards haben nur `5B` und `6B` als Analogeingang, während `5A` und `6A` dort rein digital sind. Vor dem Einsatz dieser Übung sollte daher die tatsächliche Bestückung des verwendeten Boards geprüft werden.

Zusammenfassung

Die Übung zeigt die einfachste Form der Analogwertverarbeitung auf dem DataPanel: Spannungseingang mit Hysterese einlesen und in einen numerisch nutzbaren Datentyp konvertieren. Wichtig ist dabei die Beachtung der tatsächlichen Board-Bestückung, da nicht alle DataPanel-Erweiterungsmodule dieselben Kanäle als Analogeingänge anbieten.

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

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

2.2.1.413 ISOBUS (UT, TECU, TC)

Alles rund um die ISOBUS-Kommunikation: Bedienung am Terminal, Auslesen von Traktordaten (TECU) und Anbindung an den Task Controller (TC).

2.2.1.414 Grundlagen & Logik

Hier finden Sie Übungen zu direkten Signalverbindungen, logischen Gattern (AND, OR) und grundlegenden Datenstrukturen.

2.2.1.415 Flip-Flops & Speicher

Diese Gruppe umfasst alle Übungen zur Zustandsspeicherung, von einfachen SR-Gliedern bis hin zu komplexen Klick-Mustern (Doppelklick, Long-Press).

2.2.1.416 🛠️ Übungen (Arbeitsbereich test_B)

Hier finden Sie die Dokumentation der Übungen für den logiBUS® ohne den **AX Adapter**. Die Übungen sind thematisch gruppiert, um den Lernfortschritt optimal zu unterstützen.

🎧 Podcast

- DIN EN 61499-1 Entschlüsselt: Der Bauplan für modulare, verteilte Steuerungssysteme
- FNIRSI DWS-200 Lötstation im Test: 200 Watt Power und die Wahrheit aus den Nutzer-Reviews
- 4diac IDE: Dein Open-Source-Werkzeugkasten für verteilte Industrieautomatisierung nach IEC 61499
- 4diac-Präsentation: Zielgruppen, Struktur und Alleinstellungsmerkmal Schärfen
- Den Software-Drachen zähmen: Industrielle Automatisierung und die Zukunft der Produktion

📺 Video

- Der denkende Traktor
- Der Traktor-Vordenker
- Die Kunst des Lötens
- EAGLE CAD: A Legend
- Ideale Dioden

Vollständiger Index

Falls Sie eine bestimmte Übung anhand ihrer Nummer suchen, nutzen Sie bitte die numerische Übersicht:

- Numerischer Index aller Übungen (test_B)

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

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

2.2.1.417 Schrittketten & Sequenzer

Diese Gruppe behandelt die automatische oder manuelle Abfolge von Prozessschritten, ideal für Zylindersteuerungen oder komplexe Maschinenzyklen.

2.2.1.418 Signalverarbeitung & PWM

Übungen zur Erfassung von Analogwerten, Skalierung von Messbereichen sowie zur Ansteuerung von Aktoren und LEDs mittels Pulsweitenmodulation (PWM).

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

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

2.2.1.419 Wiederverwendbare Typen (SubApps)

Hier finden Sie die Detaildokumentation der Sub-Applikations-Typen, die in den verschiedenen Übungen zur Strukturierung und Kapselung verwendet werden.

2.2.1.420 Timer & Zeitfunktionen

Hier finden Sie Übungen zu Einschalt- und Ausschaltverzögerungen, Blinkern, Impulsgebern sowie der Messung von Systemlaufzeiten.

2.2.1.421 Numerischer Index (test_B)

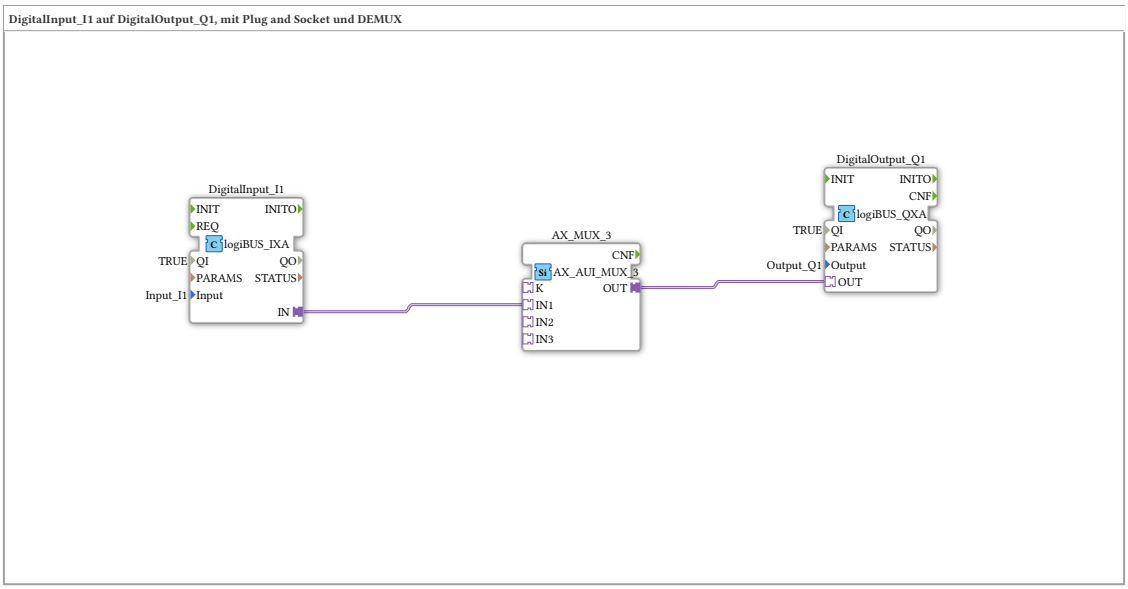
Hier finden Sie alle Übungen des Arbeitsbereichs `test_B` in aufsteigender Reihenfolge.

2.2.2 Numerischer Index (Versuche, test_B)

Hier finden Sie alle Versuche des Arbeitsbereichs `test_B` in aufsteigender Reihenfolge.

- Versuch_103c2
- Versuch_103c3
- Versuch_400
- Versuch_401
- Versuch_402
- Versuch_403
- Versuch_404
- Versuch_405
- Versuch_406
- Versuch_407
- Versuch_408
- Versuch_409

2.2.2.1 Versuch_103c2: DigitalInput auf DigitalOutput über AX-Multiplexer-Adapter



• – ○ • – ○ • – ○ •

Einleitung

Leitet einen digitalen Eingang über einen 3-fach-Multiplexer-Adapter (`AX_AUI_MUX_3`) auf einen digitalen Ausgang weiter - Plug-and-Socket-Adapterverbindung statt direkter Datenverbindung. Das dazugehörige `Uebung_103c2_AX` (test_AX) zeigt dasselbe Muster in der AX-Übungsumgebung; dieser Versuch ist die entsprechende Variante in `test_B`.

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_I1** – Typ: `logiBUS::io::DI::logiBUS_IXA`
  - Parameter: `QI = TRUE`, `Input = Input_I1`
  - Funktionsweise: digitaler Eingang, stellt seinen Zustand über den `IN`-Adapteranschluss bereit.
- **AX_MUX_3** – Typ: `adapter::selection::unidirectional::AX_AUI_MUX_3`
  - Funktionsweise: 3-fach-Multiplexer-Adapter (unidirektional) - wählt eines von bis zu drei Eingangssignalen auf einen gemeinsamen Ausgang. Hier ist nur der erste Eingang (`IN1`) beschaltet.
- **DigitalOutput_Q1** – Typ: `logiBUS::io::DQ::logiBUS_QXA`
  - Parameter: `QI = TRUE`, `Output = Output_Q1`
  - Funktionsweise: digitaler Ausgang, übernimmt seinen Zustand über den `OUT`-Adapteranschluss.

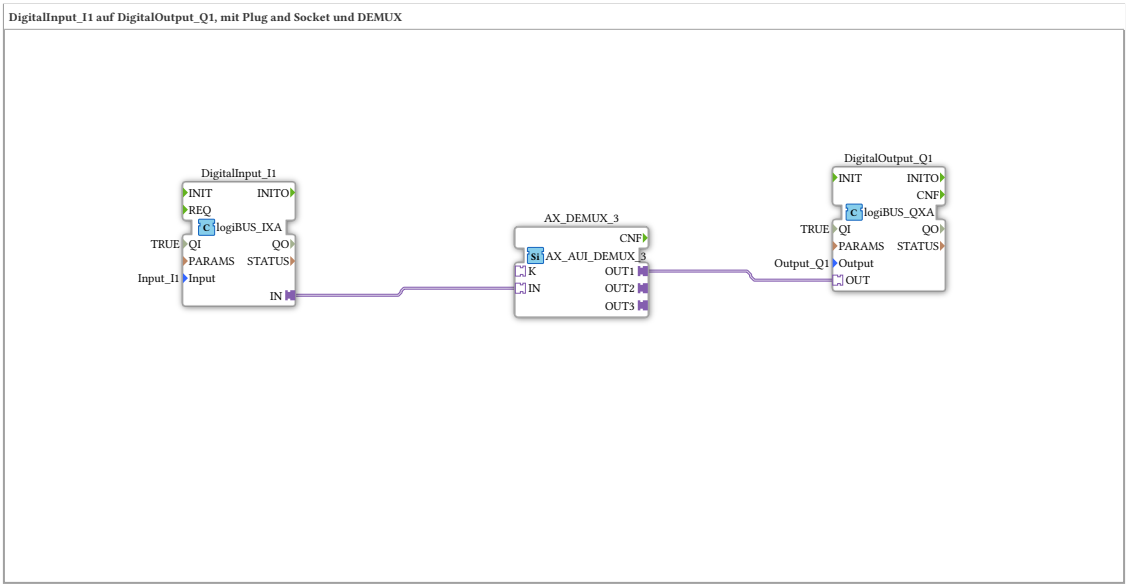
Programmablauf und Verbindungen

Adapterverbindungen (keine klassischen Daten-/Ereignisverbindungen): `DigitalInput_I1.IN` ⟶ `AX_MUX_3.IN1`, `AX_MUX_3.OUT` ⟶ `DigitalOutput_Q1.OUT`. Der physische Eingang `Input_I1` wird über den Multiplexer-Adapter durchgereicht und schaltet den physischen Ausgang `Output_Q1`.

Zusammenfassung

Zeigt die einfachste Beschaltung eines `AX_AUI_MUX_3`-Adapters mit nur einem aktiven Eingang - demonstriert die Plug-and-Socket-Adapterverbindung anstelle klassischer Datenverbindungen.

2.2.2.2 Versuch_103c3: DigitalInput auf DigitalOutput über AX-Demultiplexer-Adapter



• – ○ • – ○ • – ○ •

Einleitung

Gegenstück zu `Versuch_103c2`: statt eines Multiplexers (`AX_AUI_MUX_3`) wird hier ein 3-fach-Demultiplexer-Adapter (`AX_AUI_DEMUX_3`) verwendet, der ein Eingangssignal auf einen von mehreren möglichen Ausgängen verteilt.

Verwendete Funktionsbausteine (FBs)

- **DigitalInput_I1** – Typ: `logiBUS::io::DI::logiBUS_IXA`
  - Parameter: `QI = TRUE`, `Input = Input_I1`
- **AX_DEMUX_3** – Typ: `adapter::selection::unidirectional::AX_AUI_DEMUX_3`
  - Funktionsweise: 3-fach-Demultiplexer-Adapter (unidirektional) - verteilt ein Eingangssignal auf bis zu drei Ausgänge. Hier ist nur der erste Ausgang (`OUT1`) beschaltet.
- **DigitalOutput_Q1** – Typ: `logiBUS::io::DQ::logiBUS_QXA`
  - Parameter: `QI = TRUE`, `Output = Output_Q1`

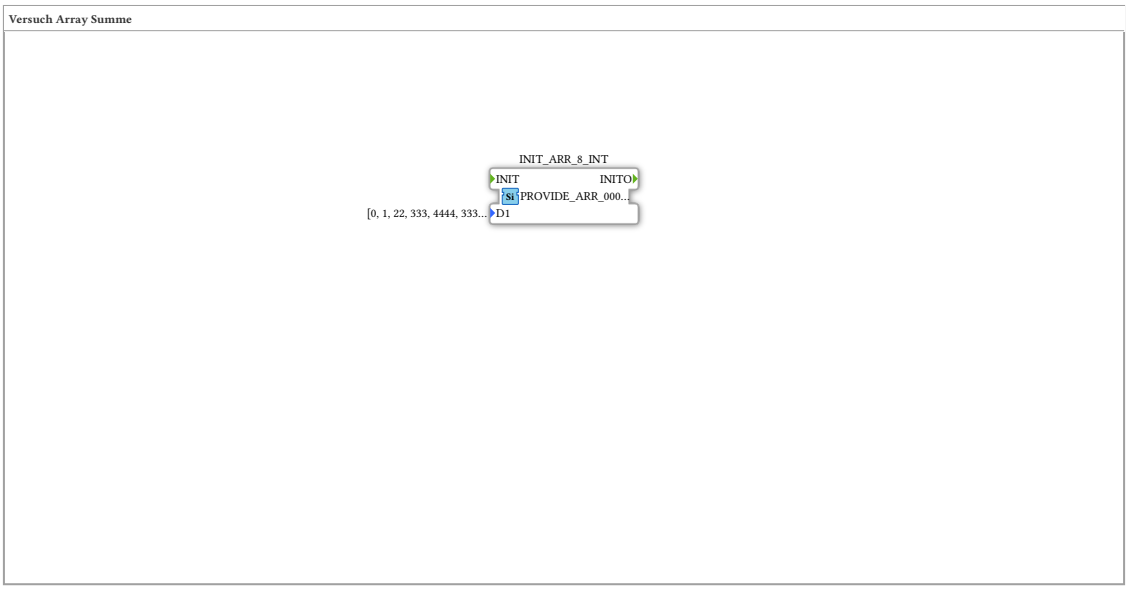
Programmablauf und Verbindungen

Adapterverbindungen: `DigitalInput_I1.IN` ⟶ `AX_DEMUX_3.IN`, `AX_DEMUX_3.OUT1` ⟶ `DigitalOutput_Q1.OUT`. Der physische Eingang `Input_I1` wird über den Demultiplexer-Adapter auf dessen ersten Ausgangszweig geleitet und schaltet den physischen Ausgang `Output_Q1`.

Zusammenfassung

Zeigt die einfachste Beschaltung eines `AX_AUI_DEMUX_3`-Adapters mit nur einem aktiven Ausgangszweig - das Gegenstück zu `Versuch_103c2`s Multiplexer-Verschaltung.

2.2.2.3 Versuch_400: Array-Konstante bereitstellen



• – ◦ • – ◦ • – ◦ •

Einleitung

Der einfachste Baustein der Array-Versuchsreihe: ein fest verdrahtetes 8-elementiges INT-Array wird bereitgestellt, aber (noch) von keinem weiteren Baustein verwendet. Dient als Ausgangspunkt für die folgenden Versuche (`Versuch_401`ff.), die auf dieser Array-Bereitstellung aufbauen.

Verwendete Funktionsbausteine (FBs)

- **INIT_ARR_8_INT** – Typ: `eclipse4diac::convert::providers::PROVIDE_ARR_0008_INT`
  - Parameter: `D1 = [0, 1, 22, 333, 4444, 333, 22, 1]`
  - Funktionsweise: stellt bei jedem Aufruf (bzw. beim FORTE-Start über sein internes `INIT`-Ereignis) das feste 8-elementige Array als Datenausgang `D1` bereit und feuert `INITO`.

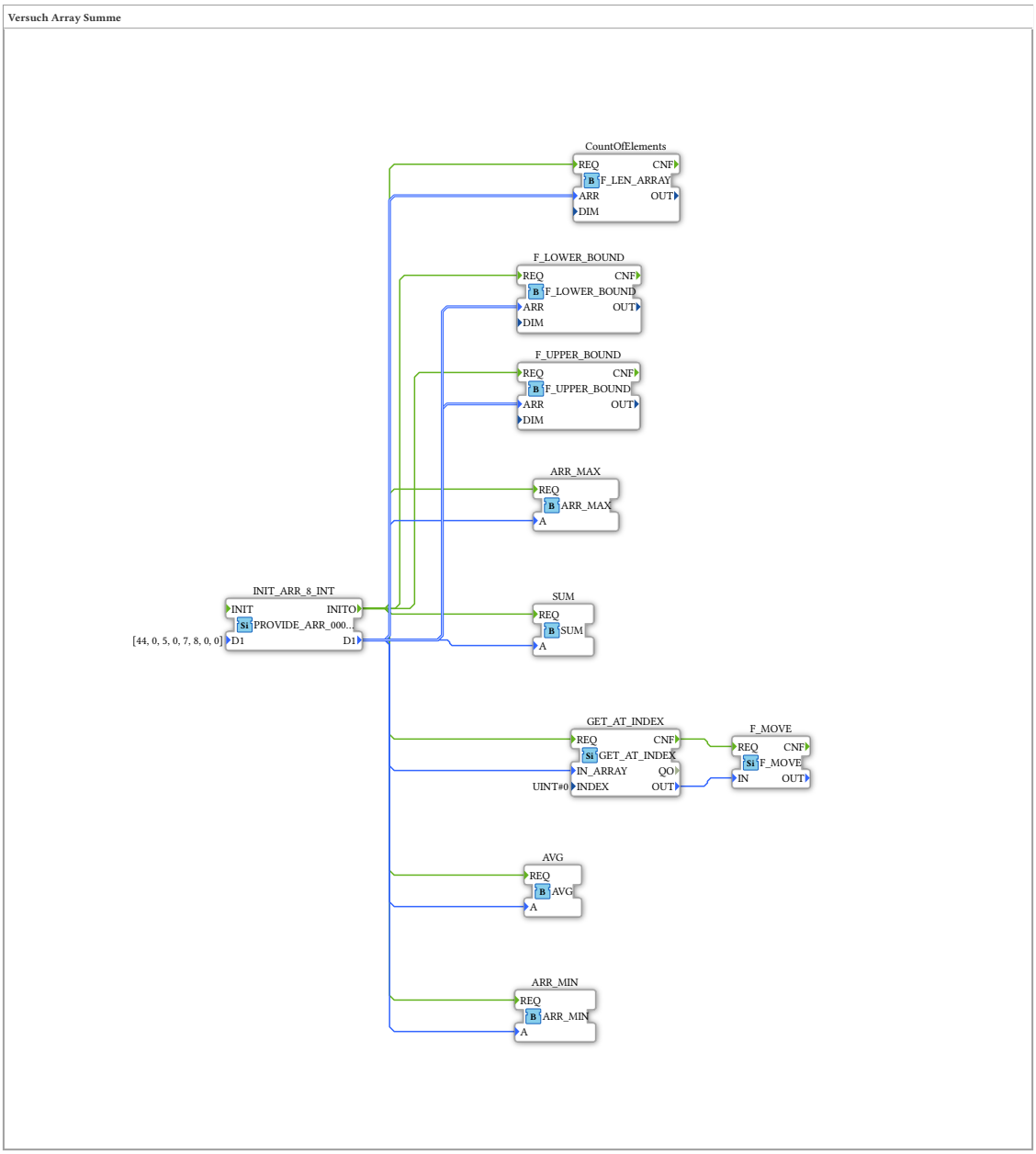
Programmablauf und Verbindungen

Es gibt nur einen Baustein und keine Verbindungen zu anderen Bausteinen - `INIT_ARR_8_INT` initialisiert sich beim Start selbst und hält seinen Array-Wert konstant am Ausgang `D1` bereit.

Zusammenfassung

Zeigt die reine Bereitstellung einer Array-Konstante über `PROVIDE_ARR_0008_INT`, ohne jede weitere Verarbeitung - Grundbaustein für die komplexeren Array-Versuche dieser Reihe.

2.2.2.4 Versuch_401: Array-Statistik mit Grenzwerten



• – ○ • – ○ • – ○ •

Einleitung

Vollständigste Ausbaustufe der Array-Statistik-Versuchsreihe: ergänzt `Versuch_406` um die Bestimmung der Array-Indexgrenzen (`F_UPPER_BOUND`/`F_LOWER_BOUND`) - insgesamt acht parallele Auswertungen desselben Arrays.

Verwendete Funktionsbausteine (FBs)

- **INIT_ARR_8_INT** – Typ: `eclipse4diac::convert::providers::PROVIDE_ARR_0008_INT`
  - Parameter: `D1 = [44, 0, 5, 0, 7, 8, 0, 0]`
- **GET_AT_INDEX** – Typ: `eclipse4diac::convert::GET_AT_INDEX`
  - Parameter: `INDEX = 0` (Element 44)
- **F_MOVE** – Typ: `iec61131::selection::F_MOVE`, Attribut: `DataType = INT`
- **ARR_MIN** / **ARR_MAX** / **AVG** / **SUM** – Typ: `logiBUS::utils::dyn_arr::*` (siehe `Versuch_406`)
- **CountOfElements** – Typ: `eclipse4diac::utils::arrays::F_LEN_ARRAY`

- **F_UPPER_BOUND** – Typ: `iec61131::arrays::F_UPPER_BOUND` – oberer Indexgrenzwert des Arrays (hier: 7).
- **F_LOWER_BOUND** – Typ: `iec61131::arrays::F_LOWER_BOUND` – unterer Indexgrenzwert des Arrays (hier: 0).

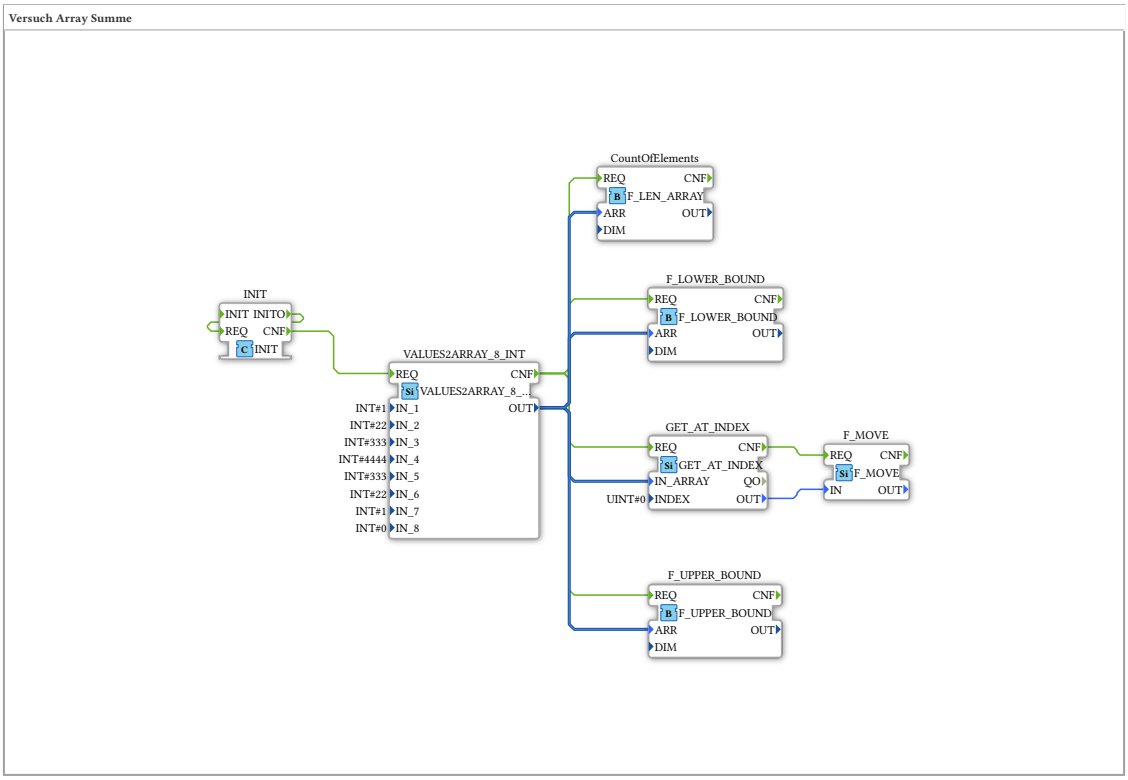
Programmablauf und Verbindungen

`INIT_ARR_8_INT.INITO` löst alle acht Konsumenten (`SUM`, `GET_AT_INDEX`, `CountOfElements`, `ARR_MAX`, `AVG`, `ARR_MIN`, `F_UPPER_BOUND`, `F_LOWER_BOUND`) gleichzeitig aus, jeder erhält dasselbe Array über `D1`. `GET_AT_INDEX` liest Index 0 (44) und gibt den Wert über `F_MOVE` weiter.

Zusammenfassung

Die umfassendste Statistik-Variante dieser Versuchsreihe: Summe, Min, Max, Mittelwert, Elementanzahl, Indexgrenzen und gezielter Indexzugriff - alle acht Auswertungen parallel aus einer einzigen Array-Quelle.

2.2.2.5 Versuch_402: Array aus Einzelwerten zusammensetzen



• – ○ • – ○ • – ○ •

Einleitung

Anders als die vorigen Versuche (feste Array-Konstante über `PROVIDE_ARR_0008_INT`) wird das Array hier aus acht einzelnen Skalarwerten zusammengesetzt (`VALUES2ARRAY_8_INT`), ausgelöst durch einen sich selbst wiederholt auslösenden `INIT`-Baustein.

Verwendete Funktionsbausteine (FBs)

- **INIT** – Typ: `iec61131::booleanOperators::INIT`
  - Funktionsweise: feuert beim Start `INITO`, das direkt auf den eigenen `REQ`-Eingang zurückgeführt ist - löst dadurch die Verarbeitungskette an.
- **VALUES2ARRAY_8_INT** – Typ: `eclipse4diac::convert::VALUES2ARRAY_8_INT`
  - Parameter: `IN_1..IN_8 = 1, 22, 333, 4444, 333, 22, 1, 0`
  - Funktionsweise: setzt aus acht einzelnen Skalarwerten ein 8-elementiges Array zusammen.
- **GET_AT_INDEX** – Typ: `eclipse4diac::convert::GET_AT_INDEX`, Parameter: `INDEX = 0` (Element 1)
- **F_MOVE** – Typ: `iec61131::selection::F_MOVE`, Attribut: `DataType = INT`

- **CountOfElements** – Typ: `eclipse4diac::utils::arrays::F_LEN_ARRAY`
- **F_UPPER_BOUND** – Typ: `iec61131::arrays::F_UPPER_BOUND`
- **F_LOWER_BOUND** – Typ: `iec61131::arrays::F_LOWER_BOUND`

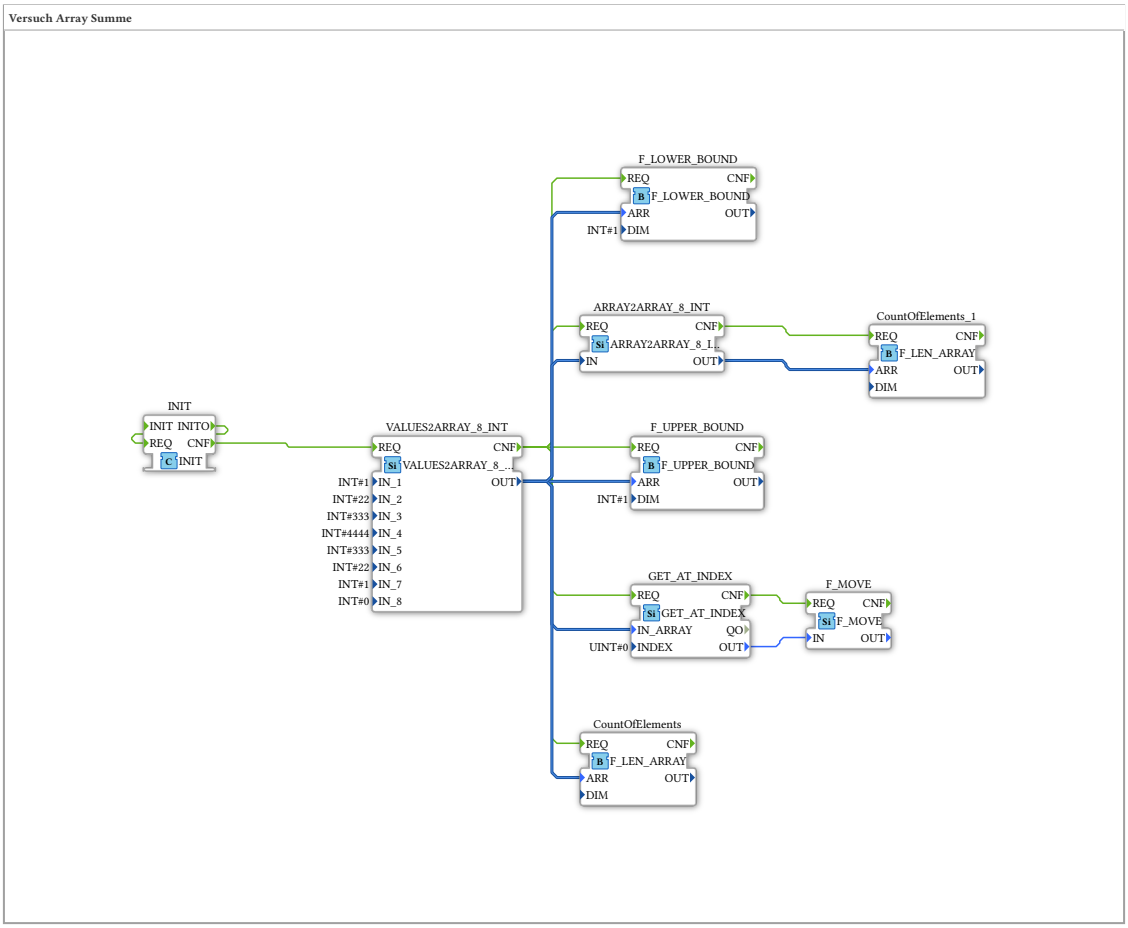
Programmablauf und Verbindungen

`INIT.CNF` löst `VALUES2ARRAY_8_INT.REQ` aus, das die acht Einzelwerte zu einem Array zusammensetzt. Dessen `CNF` löst gleichzeitig drei Konsumenten aus (`GET_AT_INDEX`, `CountOfElements`, `F_LOWER_BOUND`, `F_UPPER_BOUND`), die alle das gerade erzeugte Array über `OUT` erhalten. `GET_AT_INDEX` liest Index 0 und gibt den Wert über `F_MOVE` weiter.

Zusammenfassung

Zeigt die alternative Array-Erzeugung aus Einzelwerten (`VALUES2ARRAY_8_INT`) statt aus einer Array-Konstante, plus Auswertung von Elementanzahl und Indexgrenzen.

2.2.2.6 Versuch_403: Array kopieren und erneut auswerten



• – ○ • – ○ • – ○ •

Einleitung

Erweitert `Versuch_402` um eine Array-zu-Array-Kopie (`ARRAY2ARRAY_8_INT`): das erzeugte Array wird in eine zweite, unabhängige Array-Instanz kopiert, deren Elementanzahl separat erneut ermittelt wird.

Verwendete Funktionsbausteine (FBs)

- **INIT** – Typ: `iec61131::booleanOperators::INIT` – löst die Kette selbst aus (siehe `Versuch_402`).
- **VALUES2ARRAY_8_INT** – Typ: `eclipse4diac::convert::VALUES2ARRAY_8_INT`
  – Parameter: `IN_1..IN_8 = 1, 22, 333, 4444, 333, 22, 1, 0`
- **GET_AT_INDEX** – Typ: `eclipse4diac::convert::GET_AT_INDEX`, Parameter: `INDEX = 0`
- **F_MOVE** – Typ: `iec61131::selection::F_MOVE`, Attribut: `DataType = INT`

- **CountOfElements** – Typ: `eclipse4diac::utils::arrays::F_LEN_ARRAY` – Länge des Original-Arrays.
- **F_UPPER_BOUND** / **F_LOWER_BOUND** – Typ: `iec61131::arrays::F_UPPER_BOUND`/ `F_LOWER_BOUND`, jeweils Parameter `DIM = INT#1`.
- **ARRAY2ARRAY_8_INT** – Typ: `eclipse4diac::convert::ARRAY2ARRAY_8_INT`
  - Funktionsweise: kopiert ein 8-elementiges Array in eine neue, unabhängige Array-Instanz.
- **CountOfElements_1** – Typ: `eclipse4diac::utils::arrays::F_LEN_ARRAY` – Länge der Array-Kopie (zweite, unabhängige Instanz).

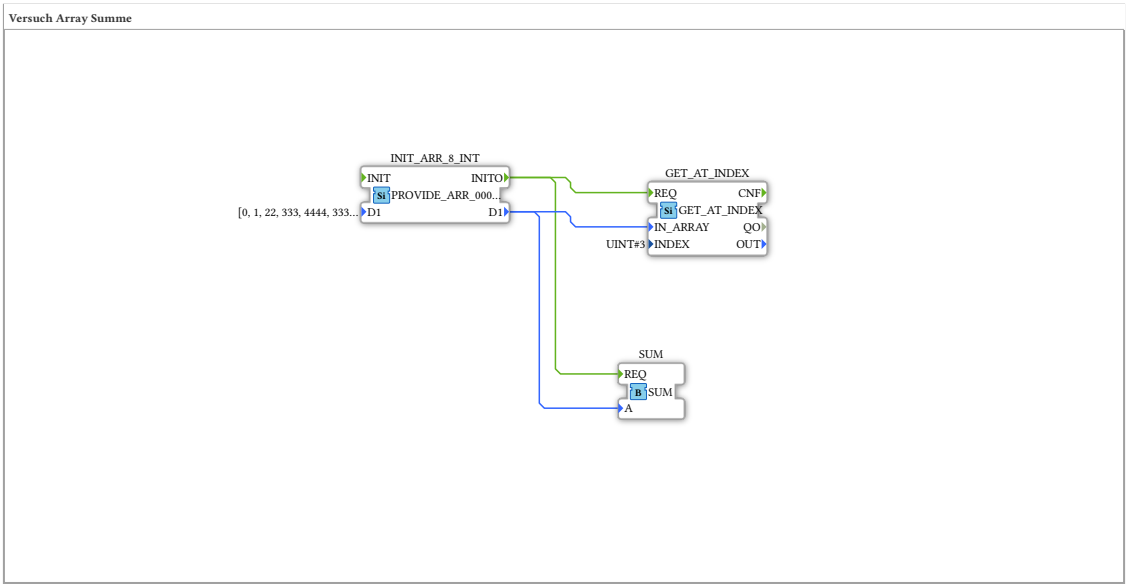
Programmablauf und Verbindungen

Wie in `Versuch_402` erzeugt `VALUES2ARRAY_8_INT.CNF` das Array und löst `GET_AT_INDEX`, `CountOfElements`, `F_LOWER_BOUND`, `F_UPPER_BOUND` sowie zusätzlich `ARRAY2ARRAY_8_INT.REQ` aus. `ARRAY2ARRAY_8_INT` kopiert das Array auf seinen eigenen `OUT`-Ausgang; dessen `CNF` löst wiederum `CountOfElements_1.REQ` aus, das die Länge der Kopie unabhängig vom Original ermittelt.

Zusammenfassung

Zeigt, wie ein Array über `ARRAY2ARRAY_8_INT` als unabhängige Kopie weiterverarbeitet werden kann, ohne das Original zu verändern - beide Instanzen (Original und Kopie) werden hier separat auf ihre Länge geprüft.

2.2.2.7 Versuch_404: Array-Summe und Einzelzugriff



• – ◦ • – ◦ • – ◦ •

Einleitung

Erweitert `Versuch_409` um einen zweiten, parallelen Konsumenten: neben der Summenbildung wird nun auch ein einzelnes Array-Element per Index gelesen.

Verwendete Funktionsbausteine (FBs)

- **INIT_ARR_8_INT** – Typ: `eclipse4diac::convert::providers::PROVIDE_ARR_0008_INT`
  - Parameter: `D1 = [0, 1, 22, 333, 4444, 333, 22, 1]`
- **GET_AT_INDEX** – Typ: `eclipse4diac::convert::GET_AT_INDEX`
  - Parameter: `INDEX = 3`
  - Eingänge: `REQ`, `IN_ARRAY`; Ausgang: `OUT`
  - Funktionsweise: liest das Element an Index 3 des Arrays (hier: `333`).
- **SUM** – Typ: `logiBUS::utils::dyn_arr::SUM`
  - Funktionsweise: summiert alle Array-Elemente.

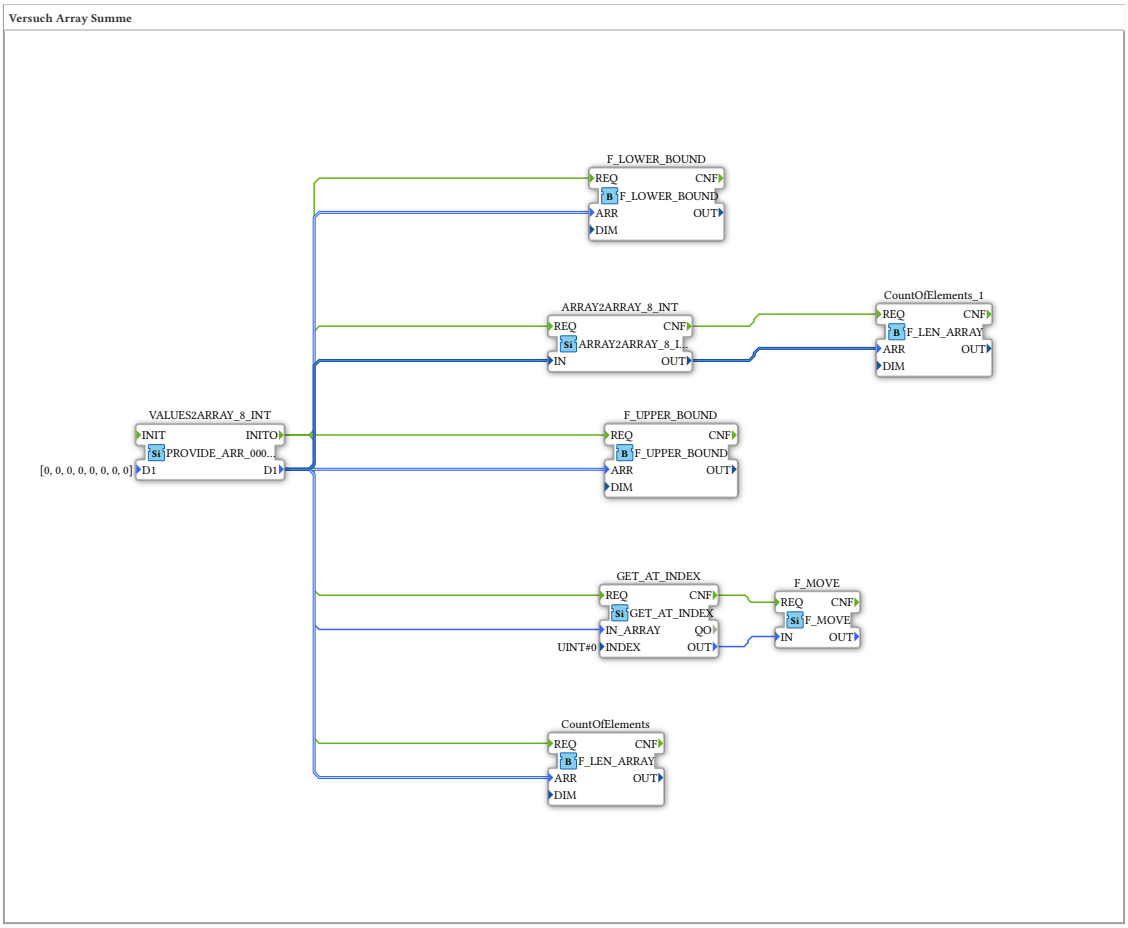
Programmablauf und Verbindungen

`INIT_ARR_8_INT.INITO` löst beide Konsumenten gleichzeitig aus: `GET_AT_INDEX.REQ` und `SUM.REQ`. Beide erhalten das gleiche Array über `D1`. `GET_AT_INDEX` liest Index 3 (`333`), `SUM` bildet die Gesamtsumme - beide Zweige laufen unabhängig voneinander parallel zueinander.

Zusammenfassung

Zeigt, dass mehrere Konsumenten dasselbe Array-Signal gleichzeitig (parallel) verarbeiten können, ohne sich gegenseitig zu beeinflussen.

2.2.2.8 Versuch_405: Array-Kopie mit Nullwerten (Konstanten-Variante von Versuch_403)



• – ◦ • – ◦ • – ◦ •

Einleitung

Strukturell identisch zu `Versuch_403` (Array-Kopie über `ARRAY2ARRAY_8_INT`, Indexzugriff, Elementanzahl, Grenzwerte), aber mit einer festen Array-Konstante (`PROVIDE_ARR_0008_INT`) statt der aus Einzelwerten zusammengesetzten Variante - direkt ausgelöst über deren eigenes `INITO`, ohne den zusätzlichen `INIT`-Baustein aus `Versuch_402`/`Versuch_403`.

Hinweis: Der Baustein trägt hier den Instanznamen `VALUES2ARRAY_8_INT`, ist aber tatsächlich vom Typ `eclipse4diac::convert::providers::PROVIDE_ARR_0008_INT` - vermutlich ein Kopfressourcen-Name aus `Versuch_402`/`Versuch_403`, der beim Bausteintyp-Wechsel nicht mehr angepasst wurde. Die folgende Beschreibung richtet sich nach dem tatsächlichen Typ.

Verwendete Funktionsbausteine (FBs)

- **VALUES2ARRAY_8_INT** (Instanzname) –
  Typ: `eclipse4diac::convert::providers::PROVIDE_ARR_0008_INT`
  – Parameter: `D1 = [0, 0, 0, 0, 0, 0, 0, 0]`

- **GET_AT_INDEX** – Typ: `eclipse4diac::convert::GET_AT_INDEX`, Parameter: `INDEX = 0`
- **F_MOVE** – Typ: `iec61131::selection::F_MOVE`, Attribut: `DataType = INT`
- **CountOfElements** – Typ: `eclipse4diac::utils::arrays::F_LEN_ARRAY`
- **F_UPPER_BOUND** / **F_LOWER_BOUND** – Typ: `iec61131::arrays::F_UPPER_BOUND`/ `F_LOWER_BOUND`
- **ARRAY2ARRAY_8_INT** – Typ: `eclipse4diac::convert::ARRAY2ARRAY_8_INT` – kopiert das Array in eine unabhängige zweite Instanz.
- **CountOfElements_1** – Typ: `eclipse4diac::utils::arrays::F_LEN_ARRAY` – Länge der Kopie.

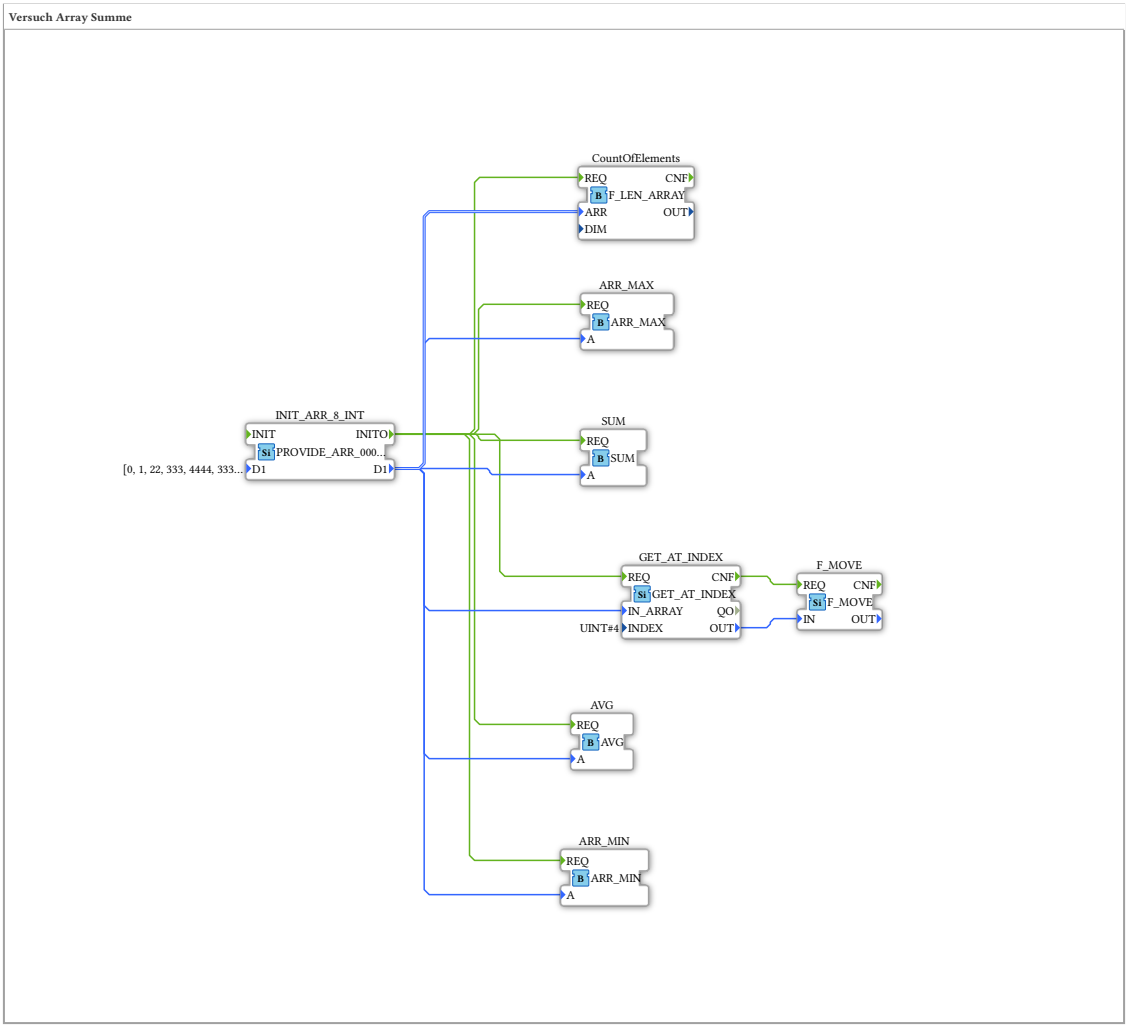
Programmablauf und Verbindungen

`INITO` löst direkt alle fünf Konsumenten gleichzeitig aus (`GET_AT_INDEX`, `CountOfElements`, `F_LOWER_BOUND`, `F_UPPER_BOUND`, `ARRAY2ARRAY_8_INT`); `ARRAY2ARRAY_8_INT.CNF` löst wiederum `CountOfElements_1.REQ` aus (Länge der Kopie).

Zusammenfassung

Gleiche Struktur wie `Versuch_403`, aber mit einer festen Array-Konstante statt einer aus Einzelwerten zusammengesetzten - und ohne den zusätzlichen `INIT`-Selbstauslöser.

2.2.2.9 Versuch_406: Array-Statistik (Summe, Min, Max, Mittelwert, Länge)



• – ○ • – ○ • – ○ •

Einleitung

Baut die parallele Auswertung eines Arrays (siehe `Versuch_404`) zu einer vollständigen Statistik aus: Summe, Minimum, Maximum, Mittelwert, Elementanzahl und ein gezielter Indexzugriff werden alle gleichzeitig aus demselben Array berechnet.

Verwendete Funktionsbausteine (FBs)

- **INIT_ARR_8_INT** – Typ: `eclipse4diac::convert::providers::PROVIDE_ARR_0008_INT`
  - Parameter: `D1 = [0, 1, 22, 333, 4444, 333, 22, 1]`
- **GET_AT_INDEX** – Typ: `eclipse4diac::convert::GET_AT_INDEX`
  - Parameter: `INDEX = 4` (Element 4444)
- **F_MOVE** – Typ: `iec61131::selection::F_MOVE`
  - Attribut: `DataType = INT`
  - Funktionsweise: übernimmt den von `GET_AT_INDEX` gelesenen Wert (reine Weitergabe/Typkonvertierung).
- **ARR_MIN** – Typ: `logiBUS::utils::dyn_arr::ARR_MIN` – ermittelt das kleinste Element.
- **ARR_MAX** – Typ: `logiBUS::utils::dyn_arr::ARR_MAX` – ermittelt das größte Element.
- **AVG** – Typ: `logiBUS::utils::dyn_arr::AVG` – berechnet den Mittelwert.
- **SUM** – Typ: `logiBUS::utils::dyn_arr::SUM` – berechnet die Summe.
- **CountOfElements** – Typ: `eclipse4diac::utils::arrays::F_LEN_ARRAY` – ermittelt die Elementanzahl.

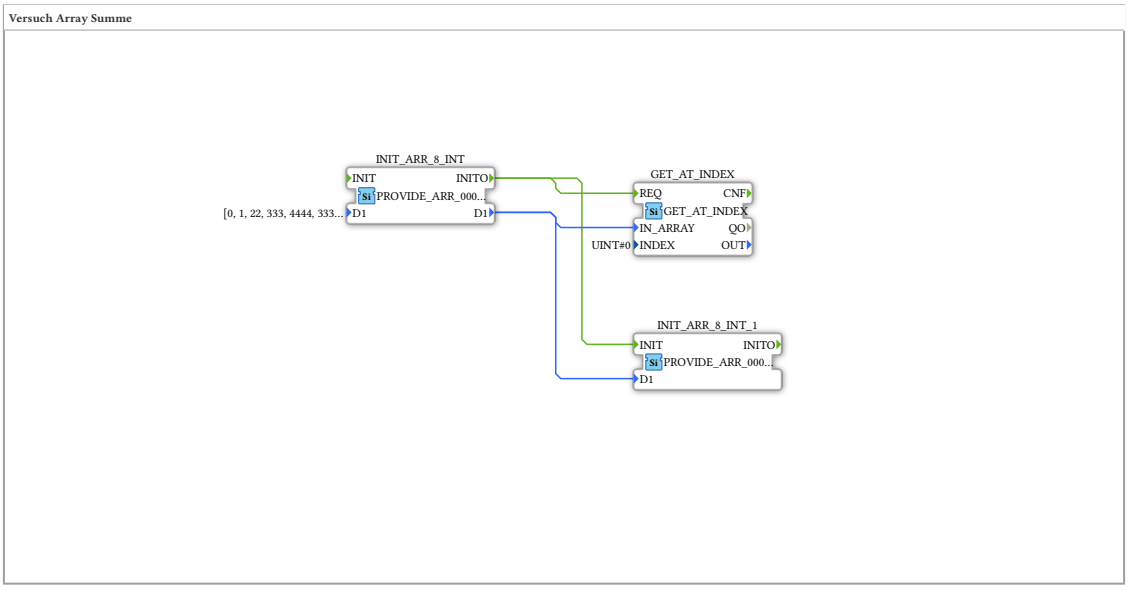
Programmablauf und Verbindungen

`INIT_ARR_8_INT.INITO` löst alle sechs Konsumenten (`SUM`, `CountOfElements`, `ARR_MAX`, `ARR_MIN`, `AVG`, `GET_AT_INDEX`) gleichzeitig aus - jeder erhält das gleiche Array über `D1`. `GET_AT_INDEX` liest Index 4 (4444) und gibt den Wert über `F_MOVE` weiter. Alle sechs Auswertungen laufen unabhängig und parallel.

Zusammenfassung

Zeigt die vollständige, parallele Standard-Statistik über ein Array (Summe/Min/Max/Mittelwert/Länge) plus gezielten Indexzugriff - Vorstufe zu `Versuch_401`, das zusätzlich die Array-Grenzen (`F_UPPER_BOUND`/`F_LOWER_BOUND`) mit einbezieht.

2.2.2.10 Versuch_407: Indexzugriff und Weiterreichen an einen zweiten Array-Provider



• – ○ • – ○ • – ○ •

Einleitung

Zeigt zwei unabhängige Verwendungen desselben Array-Signals: einen lesenden Indexzugriff und die Weiterleitung des kompletten Arrays an eine zweite `PROVIDE_ARR_0008_INT`-Instanz, die hier nicht als Konstanten-Quelle, sondern als externer angesteuerter Empfänger genutzt wird.

Verwendete Funktionsbausteine (FBs)

- **INIT_ARR_8_INT** – Typ: `eclipse4diac::convert::providers::PROVIDE_ARR_0008_INT`
  - Parameter: `D1 = [0, 1, 22, 333, 4444, 333, 22, 1]`
  - Funktionsweise: Array-Quelle, feuert beim Start `INITO`.
- **GET_AT_INDEX** – Typ: `eclipse4diac::convert::GET_AT_INDEX`
  - Parameter: `INDEX = 0` (Element `0`)
- **INIT_ARR_8_INT_1** – Typ: `eclipse4diac::convert::providers::PROVIDE_ARR_0008_INT` (zweite Instanz, ohne eigenen `D1`-Parameter)
  - Funktionsweise: erhält hier - anders als sonst in dieser Versuchsreihe üblich - Array und Ereignis von außen (`D1`/`INIT` sind hier Eingänge, die von `INIT_ARR_8_INT` gespeist werden), statt selbst als feste Konstantenquelle zu dienen.

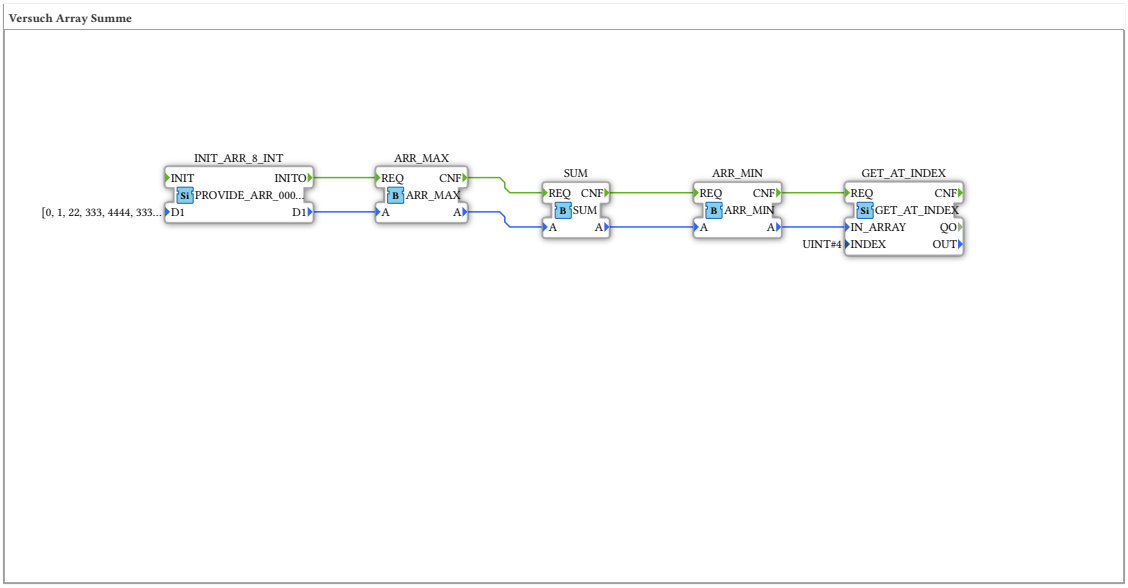
Programmablauf und Verbindungen

`INIT_ARR_8_INT.INITO` löst zwei Ziele gleichzeitig aus: `GET_AT_INDEX.REQ` (liest Index 0) und `INIT_ARR_8_INT_1.INIT` (reicht das komplette Array an die zweite Provider-Instanz weiter). Beide Datenpfade werden aus `INIT_ARR_8_INT.D1` gespeist.

Zusammenfassung

Ungewöhnliche, aber lehrreiche Variante: derselbe `PROVIDE_ARR_...`-Bausteintyp kann sowohl als reine Konstantenquelle (`INIT_ARR_8_INT`) als auch als von außen angesteuerter Empfänger (`INIT_ARR_8_INT_1`) auftreten.

2.2.2.11 Versuch_408: Array-Statistik in sequenzieller Verkettung



• – ○ • – ○ • – ○ •

Einleitung

Berechnet dieselbe Array-Statistik wie `Versuch_406` (Max, Summe, Min, Indexzugriff), aber - anders als dort - nicht parallel, sondern **sequenziell verkettet**: jeder Baustein löst erst den nächsten aus, wenn seine eigene Berechnung fertig ist (`CNF`). Zusätzlich wird das Array selbst über die `A`/`IN_ARRAY`-Datenanschlüsse von Baustein zu Baustein durchgereicht, statt allen Konsumenten gleichzeitig aus derselben Quelle zuzuführen.

Verwendete Funktionsbausteine (FBs)

- **INIT_ARR_8_INT** – Typ: `eclipse4diac::convert::providers::PROVIDE_ARR_0008_INT`
  - Parameter: `D1 = [0, 1, 22, 333, 4444, 333, 22, 1]`
- **ARR_MAX** – Typ: `logiBUS::utils::dyn_arr::ARR_MAX`
- **SUM** – Typ: `logiBUS::utils::dyn_arr::SUM`
- **ARR_MIN** – Typ: `logiBUS::utils::dyn_arr::ARR_MIN`

- **GET_AT_INDEX** – Typ: `eclipse4diac::convert::GET_AT_INDEX`, Parameter: `INDEX = 4` (Element 4444)

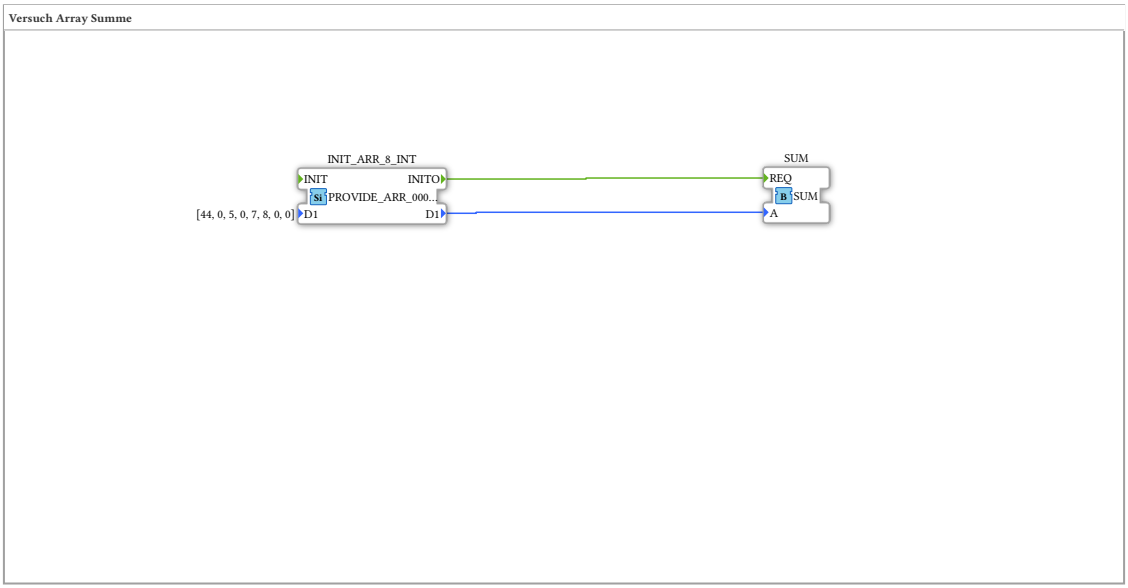
Programmablauf und Verbindungen

Ereigniskette: `INIT_ARR_8_INT.INITO` ⟶ `ARR_MAX.REQ` ⟶ (nach `ARR_MAX.CNF`) `SUM.REQ` ⟶ (nach `SUM.CNF`) `ARR_MIN.REQ` ⟶ (nach `ARR_MIN.CNF`) `GET_AT_INDEX.REQ`. Parallel dazu wird das Array selbst datentechnisch weitergereicht: `INIT_ARR_8_INT.D1` ⟶ `ARR_MAX.A` ⟶ `SUM.A` ⟶ `ARR_MIN.A` ⟶ `GET_AT_INDEX.IN_ARRAY` - jeder Baustein reicht sein eigenes `A`-Array unverändert an den nächsten weiter, statt dass alle direkt aus derselben Quelle lesen.

Zusammenfassung

Zeigt die sequenzielle (statt parallele) Verkettung von Array-Auswertungen samt Durchreichen des Array-Datenwerts von Baustein zu Baustein - ein alternatives Verdrahtungsmuster zu den parallelen Varianten `Versuch_401`/`Versuch_406`.

2.2.2.12 Versuch_409: Array-Summe



• – ◦ • – ◦ • – ◦ •

Einleitung

Erste Erweiterung von `Versuch_400`: das bereitgestellte 8-elementige Array wird nun tatsächlich verarbeitet - über die Summenbildung mittels `SUM`.

Verwendete Funktionsbausteine (FBs)

- **INIT_ARR_8_INT** – Typ: `eclipse4diac::convert::providers::PROVIDE_ARR_0008_INT`
  - Parameter: `D1 = [44, 0, 5, 0, 7, 8, 0, 0]`
  - Funktionsweise: stellt das feste 8-elementige Array bereit und feuert beim Start `INITO`.
- **SUM** – Typ: `logiBUS::utils::dyn_arr::SUM`
  - Eingänge: `REQ`, `A` (Array)
  - Funktionsweise: summiert alle Elemente des übergebenen Arrays.

Programmablauf und Verbindungen

1. **Initialisierung**: `INIT_ARR_8_INT` feuert beim Start `INITO` und stellt sein Array am Ausgang `D1` bereit.
2. **Summenbildung**: `INITO` löst `SUM.REQ` aus, `SUM` erhält das Array über `D1` ⟶ `A` und berechnet die Summe aller acht Elemente (44+0+5+0+7+8+0+0 = 64).

Zusammenfassung

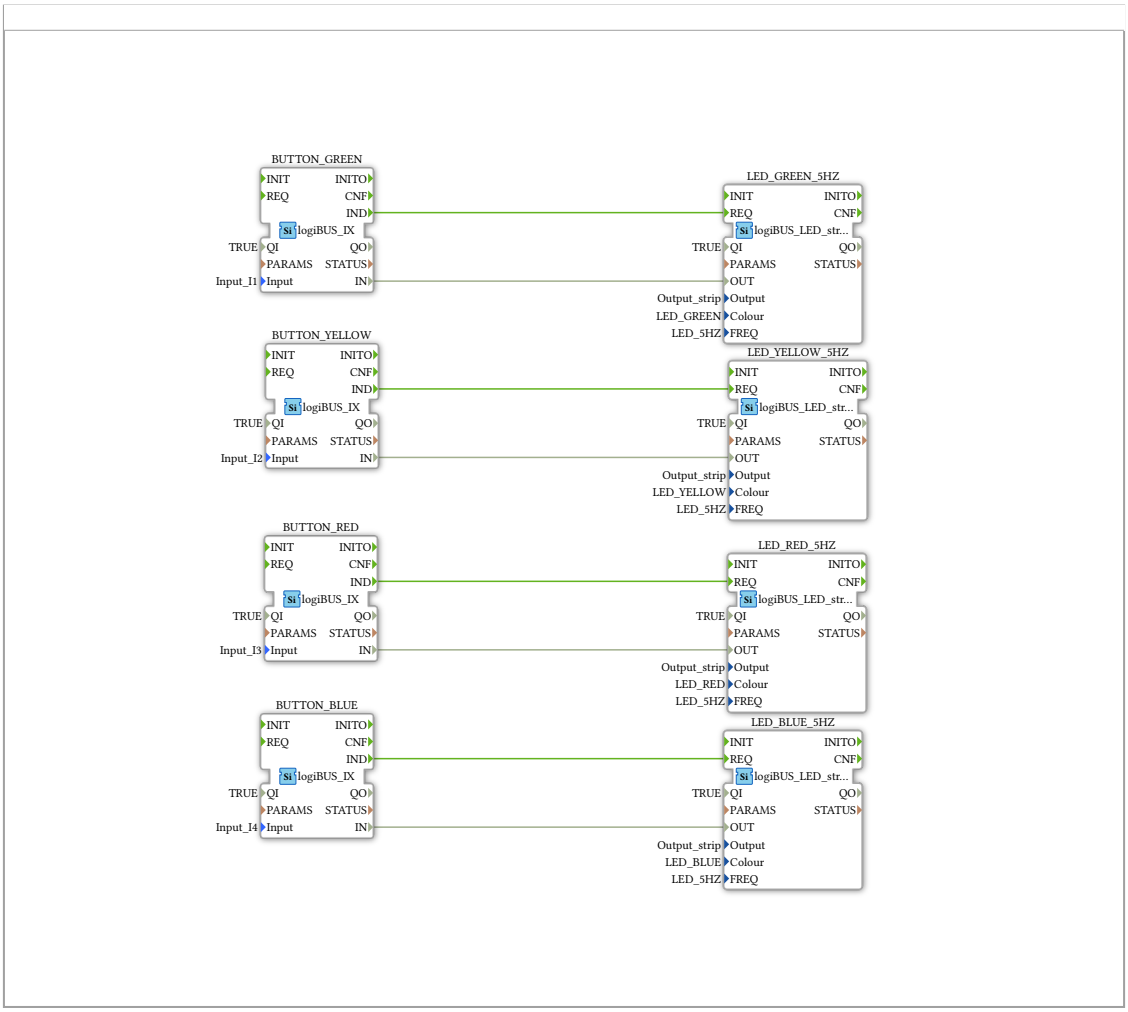
Zeigt die einfachste Form der Array-Aggregation: ein Array-Provider, ein Konsument (`SUM`). Baut direkt auf `Versuch_400` auf.

2.3 test VV

2.3.1 01 Multicast

2.3.1.1 Training 01 PUBLISH SUBSCRIBE

Training_01_PUBLISH_SUBSCRIBE: Vier verteilte Taster⟶LED-Kopplungen per Multicast



• – ○ • – ○ • – ○ •

Einleitung

`Training_01_PUBLISH_SUBSCRIBE` ist die einfachste der drei Verteilungs- Übungen unter `test_VV/sys/`: vier unabhängige Taster⟶LED-Streifen-Paare (Grün/Gelb/Rot/Blau), jeweils über ein eigenes `PUBLISH_1/SUBSCRIBE_1`- Portpaar per FORTE-eigenem Multicast auf zwei Geräte verteilt — **nicht** über OPC-UA (das zeigen `Training_01_CLIENT_SERVER`/`Training_02_OPC_UA_RES` als Alternativen für dasselbe Grundmuster). Auf dieses Multicast-Vorbild verweist `Training_02_OPC_UA_RES` selbst als „bereits gezeigt“ für den Mapping-übergreifenden Verbindungsmechanismus.

Verwendete Funktionsbausteine (FBs)

Instanz	Ort	Typ	Zweck
`BUTTON_GREEN/_YELLOW/_RED/_BLUE`	Application (`App_PUBLISH_SUBSCRIBE`)	`logiBUS::io::DI::logiBUS_IX`	Liest `Input_I1...I4`
`LED_GREEN_5HZ/_YELLOW_5HZ/_RED_5HZ/_BLUE_5HZ`	Application	`logiBUS::io::DO_LED::logiBUS_LED_Strip_QX`	Schaltet `Output_strip` in der jeweiligen Farbe, 5 Hz Blinkfrequenz

PUBLISH_BUTTON_*	Resource EMB_RES_PUBLISH (FORTE_PC_PUBLISH)	iec61499::net::PUBLISH_1	Sendet den Tasterzustand per Multicast, je Kanal ein fester Port (PORT_0…PORT_3)
SUBSCRIBE_BUTTON_*	Resource EMB_RES_SUBSCRIBE (FORTE_PC_SUBSCRIBE)	iec61499::net::SUBSCRIBE_1	Empfängt den passenden Port, treibt direkt die zugehörige LED

Die Application selbst kennt kein Netzwerk: `BUTTON_*.IND/.IN` ⟶ `LED_*.REQ/.OUT`, das rein logische Grundmuster, identisch für alle vier Farbkanäle. Erst das `Mapping` verteilt Taster auf `FORTE_PC_PUBLISH` und LEDs auf `FORTE_PC_SUBSCRIBE`.

Programmablauf und Verbindungen

1. **Application**: vier parallele, völlig unabhängige Taster⟶LED-Paare, kein Datenaustausch zwischen den Farbkanälen.
2. **Mapping**: alle vier `BUTTON_*` ⟶ `FORTE_PC_PUBLISH.EMB_RES_PUBLISH`, alle vier `LED_*_5HZ` ⟶ `FORTE_PC_SUBSCRIBE.EMB_RES_SUBSCRIBE`.
3. **Resource `EMB_RES_PUBLISH`**: `App_PUBLISH_SUBSCRIBE.BUTTON_*.IND` (dotted-path über die Mapping-Grenze) ⟶ `PUBLISH_BUTTON_*.REQ`, dazu `BUTTON_*.IN` ⟶ `PUBLISH_BUTTON_*.SD_1` — jeder Taster sendet seinen Zustand auf seinem eigenen Multicast-Port (`PORT_0` Grün … `PORT_3` Blau).
4. **Resource `EMB_RES_SUBSCRIBE`**: `SUBSCRIBE_BUTTON_*.IND` ⟶ `App_PUBLISH_SUBSCRIBE.LED_*_5HZ.REQ`, `SUBSCRIBE_BUTTON_*.RD_1` ⟶ `LED_*_5HZ.OUT` — spiegelbildlich, wieder direkte dotted-path-Verbindung über die Mapping-Grenze, ohne Zwischenbaustein.
5. **Laufzeit**: die eigentliche Übertragung zwischen `PUBLISH_BUTTON_*` und `SUBSCRIBE_BUTTON_*` existiert nicht als Modellverbindung — sie läuft ausschließlich als FORTE-Multicast über den gemeinsamen Port zur Laufzeit.

Technische Besonderheiten

- **Vier unabhängige Kanäle, ein Port je Kanal**: jedes Taster/LED-Paar bekommt seinen eigenen `PORT_n` (`VV::const::ports::myPorts`) — kein gemeinsamer Kanal, keine Verwechslungsgefahr zwischen den Farben.
- **Grundmuster für die Schwester-Übungen**: dieselbe Taster⟶LED-Struktur kehrt in `Training_01_CLIENT_SERVER` (Request/Reply statt Multicast) und `Training_02_OPC_UA_RES` (OPC-UA statt FORTE-eigenem Netzwerk) wieder — hier in der einfachsten, unidirektionalen Variante.

Lernziele

- Grundmuster FORTE-eigenes Publish/Subscribe (`PUBLISH_1`/`SUBSCRIBE_1`) als einfachste Form verteilter IEC-61499-Kommunikation.
- Mapping-übergreifende Verbindung eines Application-FB-Pins mit einem Resource-FB-Pin per dotted-path-Referenz.
- Mehrkanal-Kommunikation über feste, pro Kanal eigene Ports.

Schwierigkeitsgrad: Einfach **Vorkenntnisse**: Grundlagen Taster⟶LED-Kopplung, IEC-61499-Mapping-Konzept.

Zusammenfassung

`Training_01_PUBLISH_SUBSCRIBE` verteilt vier unabhängige Taster⟶LED-Paare per FORTE-eigenem Multicast auf zwei Geräte und dient als einfachstes Vorbild für die beiden anderen Verteilungsmuster (Request/Reply, OPC-UA) in den Schwester-Übungen dieses Verzeichnisses.

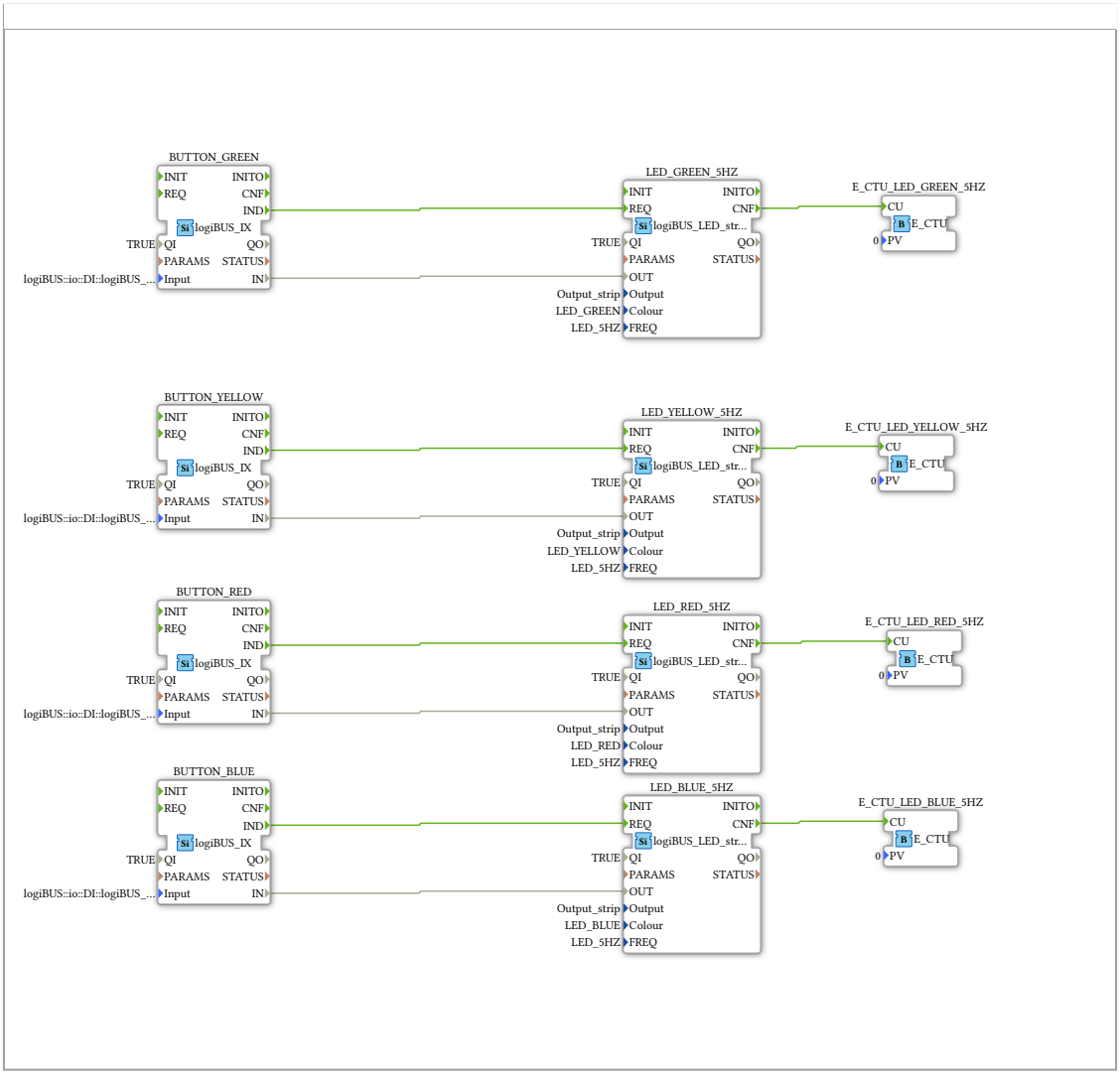
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

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

2.3.2 02 RequestReply

2.3.2.1 Training 01 CLIENT SERVER

Training_01_CLIENT_SERVER: Taster→LED per Request/Reply mit Aktivierungszähler



• — ○ • — ○ • — ○ •

Einleitung

`Training_01_CLIENT_SERVER` überträgt dasselbe Vier-Kanal-Taster→LED-Muster aus `Training_01_PUBLISH_SUBSCRIBE` auf `CLIENT_1`/`SERVER_1` (Request/ Reply) statt `PUBLISH_1`/`SUBSCRIBE_1` (Multicast). Anders als bei Multicast liefert der Server hier eine echte Antwort zurück: er zählt, wie oft jede LED aktiviert wurde, und meldet den aktuellen Zählerstand an den Client zurück.

Verwendete Funktionsbausteine (FBs)

Instanz	Ort	Typ	Zweck
`BUTTON_GREEN/ _YELLOW/_RED/_BLUE`	Application (`App_CLIENT_SERVER`)	`logiBUS::io::DI::logiBUS_IX`	Liest `Input_I1...I4`
`LED_GREEN_5HZ/ _YELLOW_5HZ/ _RED_5HZ/_BLUE_5HZ`	Application	`logiBUS::io::DO_LED::logiBUS_LED_Strip_8CH`	Schreibt `Output_strip`, 5 Hz Blinkfrequenz

CLIENT_BUTTON_*	Resource EMB_RES_CLIENT (FORTE_PC_CLIENT)	iec61499::net::CLIENT_1	Sendet die Aktivierungsanfrage, empfängt den Zählerstand als Antwort
UINT2UINT_0..._3	Resource EMB_RES_CLIENT	iec61131::selection::F_MOVE (UINT)	Nimmt die Antwort (RD_1) entgegen — Endpunkt ohne weitere Verwendung im Modell
SERVER_BUTTON_*	Resource EMB_RES_SERVER (FORTE_PC_SERVER)	iec61499::net::SERVER_1	Empfängt die Anfrage, schaltet die LED, sendet den Zählerstand zurück
E_CTU_LED_*_5HZ	Application, gemappt auf EMB_RES_SERVER	iec61499::events::E_CTU	Zählt die Aktivierungen je Farbe (CU bei jeder LED.CNF)

Programmablauf und Verbindungen

1. **Application**: vier unabhängige Taster→LED-Paare, strukturell identisch zu Training_01_PUBLISH_SUBSCRIBE; zusätzlich je ein E_CTU-Zähler pro Farbe, der auf LED_*_5HZ.CNF reagiert.
2. **Mapping**: alle vier BUTTON_* → FORTE_PC_CLIENT.EMB_RES_CLIENT, alle vier LED_*_5HZ und E_CTU_LED_*_5HZ → FORTE_PC_SERVER.EMB_RES_SERVER.
3. **Resource EMB_RES_CLIENT**: App_CLIENT_SERVER.BUTTON_*.IND (dotted- path) → CLIENT_BUTTON_*.REQ, BUTTON_*.IN → CLIENT_BUTTON_*.SD_1 — der Taster löst die Anfrage samt Zustand aus, jeder Kanal auf seinem eigenen Port (PORT_B0…PORT_B3). Die Antwort (CLIENT_BUTTON_*.CNF und .RD_1) läuft in UINT2UINT_* — ein reiner Endpunkt ohne Weiterverarbeitung im Modell.
4. **Resource EMB_RES_SERVER**: SERVER_BUTTON_*.IND → App_CLIENT_SERVER.LED_*_5HZ.REQ, SERVER_BUTTON_*.RD_1 → LED_*_5HZ.OUT — die Anfrage schaltet die LED direkt. Deren Bestätigung (LED_*_5HZ.CNF) zählt E_CTU_LED_*_5HZ.CU hoch; dessen CUO-Ereignis löst SERVER_BUTTON_*.RSP aus, mit dem aktuellen Zählerstand (E_CTU.CV) als SD_1 — die eigentliche Antwort geht also erst raus, wenn die LED tatsächlich bestätigt hat.

Technische Besonderheiten

- **Echte, zustandsbehaftete Antwort statt Dummy-Wert**: anders als in der Schwester-Übung Training_02_CLIENT_SERVER2 (dort sendet der Server nur einen konstanten Platzhalterwert zurück) trägt die Antwort hier den tatsächlichen, laufend aktualisierten Aktivierungszähler je Kanal.
- **Antwort erst nach Bestätigung**: SERVER_BUTTON_*.RSP wird nicht direkt von SERVER_BUTTON_*.IND ausgelöst, sondern erst indirekt über LED_*_5HZ.CNF → E_CTU.CUO — die Antwort bestätigt also, dass die LED wirklich reagiert hat, nicht nur, dass die Anfrage angekommen ist.
- **Client verwirft die Antwort aktuell**: UINT2UINT_* (F_MOVE) nimmt den Zählerstand zwar entgegen, hat aber keinen weiteren Ausgang im Modell — ein Ansatzpunkt für eine Erweiterung (z. B. Anzeige des Zählerstands), nicht bereits Teil dieser Übung.

Lernziele

- Grundmuster FORTE-eigenes Request/Reply (CLIENT_1/SERVER_1) im Vergleich zum reinen Publish/Subscribe aus Training_01_PUBLISH_SUBSCRIBE.
- Antwortdaten zeitlich von der eigentlichen Aktion entkoppeln (Antwort erst nach Bestätigung, nicht sofort bei Anfrageeingang).
- Zählerbaustein (E_CTU) als Zustand zwischen mehreren Ereignissen.

Schwierigkeitsgrad: Mittel **Vorkenntnisse**: Training_01_PUBLISH_SUBSCRIBE (Grundmuster Taster→LED verteilt), IEC-61499-Mapping-Konzept.

Zusammenfassung

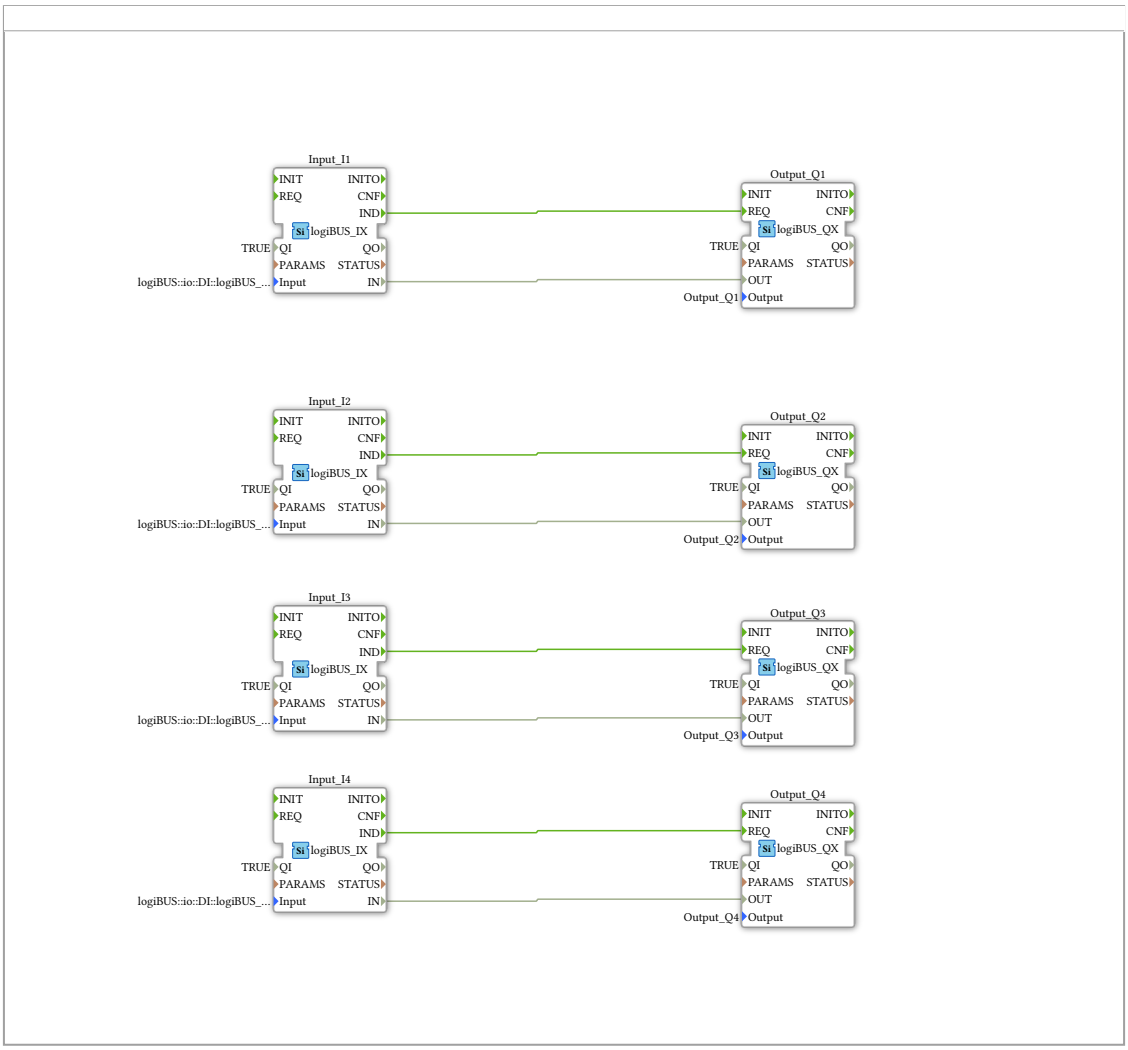
`Training_01_CLIENT_SERVER` zeigt Request/Reply als Alternative zu Multicast für dasselbe Vier-Kanal-Muster: der Server bestätigt jede Aktivierung nicht nur, sondern liefert einen echten, mitgezählten Wert als Antwort zurück, statt nur einen Platzhalter zu senden.

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

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

2.3.2.2 Training 02 CLIENT SERVER2

Training_02_CLIENT_SERVER2: Reduziertes Request/Reply mit einfachem I/O



• — ○ • — ○ • — ○ •

Einleitung

`Training_02_CLIENT_SERVER2` ist eine reduzierte Variante von `Training_01_CLIENT_SERVER`: dieselbe Vier-Kanal-Request/Reply-Struktur (`CLIENT_1`/`SERVER_1`), aber mit einfachen digitalen Ein-/Ausgängen (`logiBUS_IX`/`logiBUS_QX`) statt Tastern mit blinkenden LED-Streifen, und ohne Aktivierungszähler auf der Server-Seite — die Antwort ist hier ein fester Platzhalterwert (`DINT#0`), nicht der echte Zählerstand aus `Training_01_CLIENT_SERVER`. Zeigt damit das Request/Reply-Protokoll in seiner minimalen Form, ohne die zusätzliche Zähler-Logik.

Verwendete Funktionsbausteine (FBs)

Instanz	Ort	Typ	Zweck
`Input_I1..._I4`	Application (`App_CLIENT_SERVER`)	`logiBUS::io::DI::logiBUS_IX`	Liest `Input_I1...I4`
`Output_Q1..._Q4`	Application	`logiBUS::io::DQ::logiBUS_QX`	Schaltet `Output_Q1...Q4`
`CLIENT_BUTTON_*`	Resource `EMB_RES_CLIENT` (`FORTE_PC_CLIENT`)	`iec61499::net::CLIENT_1`	Sendet die Anfrage samt Eingangszustand
`UINT2UINT_0..._3`	Resource `EMB_RES_CLIENT`	`iec61131::selection::F_MOVE` (UINT)	Nimmt die Antwort entgegen — Endpunkt ohne weitere Verwendung
`SERVER_BUTTON_*`	Resource `EMB_RES_SERVER` (`FORTE_PC_SERVER`)	`iec61499::net::SERVER_1`	Empfängt die Anfrage, schaltet den Ausgang direkt, antwortet mit `DINT#0`

Programmablauf und Verbindungen

1. **Application**: vier unabhängige `Input_I*`→`Output_Q*`-Paare, ohne Zähler oder Blinklogik — das einfachste I/O-Grundmuster.
2. **Mapping**: alle vier `Input_I*` → `FORTE_PC_CLIENT.EMB_RES_CLIENT`, alle vier `Output_Q*` → `FORTE_PC_SERVER.EMB_RES_SERVER`.
3. **Resource `EMB_RES_CLIENT`**: `Input_I*.IND`/`.IN` (dotted-path) → `CLIENT_BUTTON_*.REQ`/`.SD_1`, je Kanal ein eigener Port (`PORT_B0`...`PORT_B3`). Die Antwort läuft wie in `Training_01_CLIENT_SERVER` in `UINT2UINT_*` — unverändert übernommenes Muster, weiterhin ein Endpunkt ohne Weiterverarbeitung.
4. **Resource `EMB_RES_SERVER`**: `SERVER_BUTTON_*.IND`/`.RD_1` → `Output_Q*.REQ`/`.OUT` — **direkt**, ohne Zähler oder Umweg über eine Bestätigung. `SERVER_BUTTON_*` hat als `SD_1`-Parameter fest `DINT#0` gesetzt: die Antwort trägt immer denselben Platzhalterwert, unabhängig vom tatsächlichen Schaltvorgang.

Technische Besonderheiten

- **Statische statt gezählte Antwort**: der Parameter `SD_1="DINT#0"` an `SERVER_BUTTON_*` ersetzt das `E_CTU`-Zählmuster aus `Training_01_CLIENT_SERVER` — die Antwort trägt keine echte Information mehr, nur noch die Protokoll-Bestätigung selbst zählt.
- **Keine Bestätigungskette**: anders als in `Training_01_CLIENT_SERVER` (Antwort erst nach `LED.CNF`) löst hier `SERVER_BUTTON_*.IND` die Antwort indirekt gar nicht erst aus — der Ausgang wird sofort geschaltet, ohne Zwischenschritt.
- **Gleiches Client-seitiges Muster wie in `Training_01_CLIENT_SERVER`**: die `UINT2UINT_*`-Blöcke (dead-end-Antwortempfang) sind identisch übernommen — die Vereinfachung betrifft ausschließlich die Server-Seite und die I/O-Bausteine.

Lernziele

- Request/Reply-Protokoll (`CLIENT_1`/`SERVER_1`) in seiner minimalen Form, ohne zusätzliche Zustands-/Zähllogik.
- Unterschied zwischen einer inhaltlich bedeutungsvollen Antwort (`Training_01_CLIENT_SERVER`) und einer reinen Protokoll-Bestätigung mit Platzhalterwert.

Schwierigkeitsgrad: Einfach **Vorkenntnisse**: `Training_01_CLIENT_SERVER` (vollständiges Request/Reply- Muster mit Zähler) als Vergleichspunkt.

Zusammenfassung

`Training_02_CLIENT_SERVER2` reduziert `Training_01_CLIENT_SERVER` auf das Nötigste: einfache digitale I/O-Kanäle, direkte Ausgangsschaltung ohne Umweg, und eine Antwort, die nur noch als Platzhalter dient statt einen echten Zählerstand zu tragen.

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

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

2.3.3 03 OPC UA

2.3.3.1 OPC-UA-Adressen (`myOpcUaAddresses.gcf`)

Die OPC-UA-IDs für die Trainings `test_VV/sys/03_OPC_UA/Training_01` bis `_12` werden in der GCF aus wenigen Grundwerten berechnet. Die Functions dafür (`F_OPC_LOCAL`, `F_OPC_REMOTE`, `F_MGR_TO_OPC`) stehen in der Lib `MyLib_OPC`. Der 4diac-Export wertet sie aus, im Bootfile stehen die fertigen Strings.

Stationen ändern: nur hier

```
PC_A : WSTRING := logiBUS::stations::Computers::C_11;
PC_B : WSTRING := logiBUS::stations::Computers::C_12;
```

- Die `.sys`-Dateien nutzen `PC_A`/`PC_B` als `MGR_ID` und importieren `VV::const::OPC_UA::myOpcUaAddresses::PC_A`/`PC_B`.
- Die OPC-UA-Endpunkte `PC_A_OPC`/`PC_B_OPC` werden daraus mit `MyLib_OPC::F_MGR_TO_OPC` abgeleitet (`192.168.1.12:61499` -> `opc.tcp://192.168.1.12:4840`).

Eine andere Station, etwa `C_13`, wirkt damit auf Deployment und alle Remote-IDs.

Aufbau pro Signal

```
Q1_NODE         := "/Objects/DigitalOutput/Q1,1:s=Output_Q1";           // Knoten,
einmal
Q1_LOCAL_READ   := MyLib_OPC::F_OPC_LOCAL("READ", Q1_NODE);              // Station,
die den Knoten hat
Q1_REMOTE_WRITE := MyLib_OPC::F_OPC_REMOTE("WRITE", PC_B_OPC, Q1_NODE);   // andere
Station
```

Function	Ergebnis
`F_OPC_LOCAL(action, node)`	`opc_ua[ACTION;NODE]`
`F_OPC_REMOTE(action, endpoint, node)`	`opc_ua[ACTION;ENDPOINT#;NODE]`

Zusammengehörige Aktionen (siehe 4diac-Doku `opcua.adoc`):

Lokal	passender FB lokal	Remote	passender FB remote
`READ`	`AX_SUBSCRIBE_1` / `SUBSCRIBE_n`	`WRITE`	`CLIENT_n_0`
`CREATE_METHOD`	`SERVER_0`	`CALL_METHOD`	`CLIENT_0`

- Der Name-Suffix `*_LOCAL_*`/`*_REMOTE_*` ist geblieben, damit die SubApps unverändert weiterlaufen.
- `SW2_*` bündelt zwei Knoten in einem String (mit `;` getrennt) für `SUBSCRIBE_2`/`CLIENT_2_0`.

Hinweise

- `Comment=` in der GCF kurz halten, sonst schneidet der Resave der IDE lange Kommentare ab. Ausführliches gehört in diese Datei.
- Geprüft: Die Bootfiles von Training_01 (PC_A und PC_B) sind vor und nach der Umstellung byte-identisch.

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

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

2.3.3.2 Training 01 OPC UA SUB

Training_01_OPC_UA_SUB: Verteiltes I1→Q1 über OPC-UA („SUB style“)



• – ◦ • – ◦ • – ◦ •

Einleitung

`Training_01_OPC_UA_SUB` zeigt denselben Anwendungsfall wie `Training_02_OPC_UA_RES` (`Input_I1` auf Gerät A → `Output_Q1` auf Gerät B, über OPC-UA verteilt) — aber als **Gegenentwurf**: den **„SUB style“**. Statt die Kommunikationsbausteine in der `<Resource>` jedes Geräts zu platzieren, steckt das OPC-UA-Protokoll hier direkt in einem wiederverwendbaren `MyLib::sys`-Composite. Namensgebend ist „SUB“ für `subapp::Training_01_OPC_UA_SUB`, das Package, in dem beide Geräte-Composites liegen. Laut Kommentar in den Quellbausteinen entspricht dieses Muster einem **echten Produktivmuster** („Krauternter“) aus der Praxis.

Verwendete Funktionsbausteine (FBs)

Instanz	Ort	Typ	Zweck
`SubApp_PC_A`	Application (`App_OPC_UA_SUB`)	`subapp::Training_01_OPC_UA_SUB::SubApp_PC_A`	Kapselt Gerät-A-seitige Logik komplett
`SubApp_PC_B`	Application	`subapp::Training_01_OPC_UA_SUB::SubApp_PC_B`	Kapselt Gerät-B-seitige Logik komplett
`INPUT_I1` (in `SubApp_PC_A`)	—	`MyLib::sys::logiBUS_IXA_TO_CLIENT_OPC`	Liest `Input_I1` und schreibt aktiv per OPC-UA (`CLIENT_1_0`) in einem Baustein
`OUTPUT_Q1` (in `SubApp_PC_B`)	—	`MyLib::sys::logiBUS_QXA_FROM_SUBSCRIBE_OPC`	Schreibt `Output_Q1` basierend auf einem lokal überwachten OPC-UA-Knoten (`SUBSCRIBE_1`) in einem Baustein

Beide Geräte-Resources (`EMB_RES_A`, `EMB_RES_B`) enthalten **nichts** außer einem generischen `E_TRIG` (`EVENTTYPE='EInit'`) — anders als bei `Training_02_OPC_UA_RES`, wo `CLIENT_Q1`/`SUBSCRIBE_Q1` explizit in der Resource stehen. Das gesamte Protokollwissen ist hier in `logiBUS_IXA_TO_CLIENT_OPC`/`logiBUS_QXA_FROM_SUBSCRIBE_OPC` gekapselt.

OPC-UA-Adressraum

Die Stationen `PC_A`/`PC_B` (`MGR_ID` der Geräte) stehen nur in `myOpcUaAddresses.gcf`; die OPC-UA-IDs berechnet die GCF mit den Functions der Lib `MyLib_OPC` (`F_OPC_LOCAL`, `F_OPC_REMOTE`, `F_MGR_TO_OPC`). Die Namen `*_LOCAL_*`/`*_REMOTE_*` bleiben dabei erhalten. Aufbau und Berechnung: OPC-UA-Adressen.

Dieselben Konstanten wie in `Training_02_OPC_UA_RES`, aus `VV::const::OPC_UA::myOpcUaAddresses`:

Konstante	Verwendet von
`Q1_REMOTE_WRITE`	`SubApp_PC_A.INPUT_I1` (ID-Parameter)
`Q1_LOCAL_READ`	`SubApp_PC_B.OUTPUT_Q1` (ID-Parameter)

Programmablauf und Verbindungen

1. **Application** (`App_OPC_UA_SUB`): instanziiert `SubApp_PC_A` und `SubApp_PC_B` direkt, **ohne** jede Verbindung zwischen ihnen — die gesamte Kopplung läuft ausschließlich als OPC-UA-Kommunikation zur Laufzeit, es gibt keine Modellverbindung zwischen den beiden Composites.
2. **`SubApp_PC_A`**: eine einzige Instanz `INPUT_I1` (`logiBUS_IXA_TO_CLIENT_OPC`, `Input=Input_I1`, `ID=Q1_REMOTE_WRITE`) — liest den physischen Eingang und schreibt seinen Zustand aktiv per OPC-UA remote, alles in einem Composite-Baustein.
3. **`SubApp_PC_B`**: spiegelbildlich eine einzige Instanz `OUTPUT_Q1` (`logiBUS_QXA_FROM_SUBSCRIBE_OPC`, `Output=Output_Q1`, `ID=Q1_LOCAL_READ`) — überwacht lokal den von Gerät A beschriebenen Knoten und schaltet den physischen Ausgang, ebenfalls in einem Composite-Baustein.
4. **Mapping**: `SubApp_PC_A` → `FORTE_PC_A.EMB_RES_A`, `SubApp_PC_B` → `FORTE_PC_B.EMB_RES_B`. Die Resources selbst tragen keine fachliche Logik — nur der generische `EInit`-Trigger, den 4diac ohnehin für jede Resource vorsieht.

Technische Besonderheiten

- **Protokoll im Composite statt in der Resource**: der komplette Unterschied zu `Training_02_OPC_UA_RES` liegt darin, WO das OPC-UA-Wissen steckt — hier in einem einzigen wiederverwendbaren `MyLib::sys`-Baustein pro Geräteseite, dort verteilt auf Application-FB + separaten Resource-Adapterbaustein. Funktional identisch, aber die „SUB style"- Variante ist als fertiger Baustein direkt wiederverwendbar, ohne dass jedes neue Projekt die Resource-Verdrahtung erneut nachbauen muss.
- **Reales Produktivmuster**: laut Kommentar in `SubApp_PC_A`/`SubApp_PC_B` ist dies kein Lehrbeispiel, sondern ein in der Praxis eingesetztes Muster („Krauternter").
- **Leere Resources**: da die gesamte Logik im Composite steckt, bleibt in beiden Resources nur der obligatorische `EInit`-Trigger übrig — ein deutlicher struktureller Kontrast zur RES-Variante.

Lernziele

- Zwei gleichwertige Architekturmuster für dieselbe verteilte OPC-UA- Aufgabe: Protokoll in der Resource („RES style") vs. Protokoll im wiederverwendbaren Composite („SUB style").
- Vor-/Nachteile der Kapselung: Wiederverwendbarkeit des fertigen Composite-Bausteins gegen Transparenz der einzelnen Protokollschritte in der Resource.
- Bezug zu einem echten Produktivmuster statt eines rein didaktischen Beispiels.

Schwierigkeitsgrad: Mittel **Vorkenntnisse**: `Training_02_OPC_UA_RES` (Gegenstück im „RES style"), Grundlagen der OPC-UA-Adapterbausteine.

Zusammenfassung

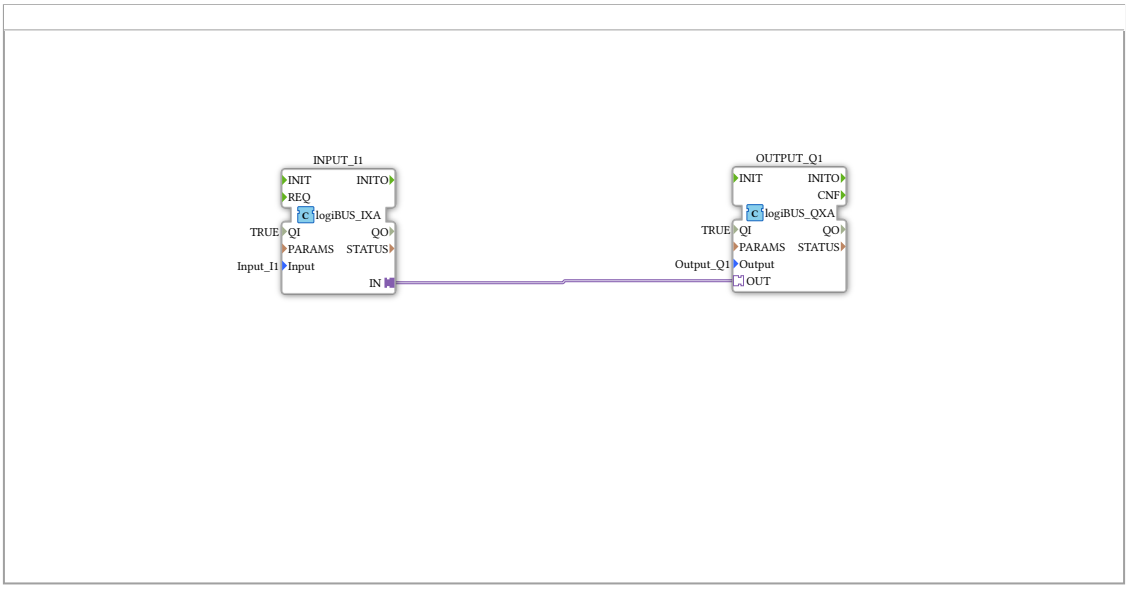
`Training_01_OPC_UA_SUB` löst dieselbe Verteilungsaufgabe wie `Training_02_OPC_UA_RES`, verlagert das OPC-UA-Protokollwissen aber komplett in wiederverwendbare `MyLib::sys`-Composites statt es in der Resource jedes Geräts zu verdrahten — ein reales, in der Praxis eingesetztes Architekturmuster.

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

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

2.3.3.3 Training 02 OPC UA RES

Training_02_OPC_UA_RES: Verteiltes I1→Q1 über OPC-UA („RES style“)



• – ◦ • – ◦ • – ◦ •

Einleitung

`Training_02_OPC_UA_RES` zeigt das Grundmuster `Input_I1` ⟶ `Output_Q1` aus `test_B/Uebungen/Uebung_001.SUB` — jetzt aber echt **verteilt** auf zwei FORTE-Geräte: `FORTE_PC_A` (192.168.1.11) liest `Input_I1`, `FORTE_PC_B` (192.168.1.12) schaltet `Output_Q1`. Die Verbindung zwischen beiden läuft über OPC-UA, nicht über FORTE-eigenes Multicast (das zeigt bereits `Training_01_PUBLISH_SUBSCRIBE`).

Namensgebend ist der **„RES style“**: die eigentlichen Kommunikationsbausteine (`AX_CLIENT_1_0`, `AX_SUBSCRIBE_1`) stecken in der `<Resource>` jedes Geräts, nicht im Application-Composite selbst. Das entspricht dem kanonischen IEC-61499-Verteilungsmuster aus `distribute4diac.adoc` sowie dem bestehenden `Training_01_PUBLISH_SUBSCRIBE`- Vorbild. Die Schwester-Übung `Training_01_OPC_UA_SUB` zeigt für denselben Anwendungsfall den Gegenentwurf („SUB style“, Krauternter-Produktivmuster): dort steckt das Protokoll direkt im wiederverwendbaren `MyLib::sys`- Composite statt in der Resource.

Verwendete Funktionsbausteine (FBs)

Instanz	Ort	Typ	Zweck
`INPUT_I1`	Application (`App_OPC_UA_RES2`)	`logiBUS::io::DI::logiBUS_IXA`	Liest `Input_I1`, stellt den Zustand als AX-Adapter-Plug bereit
`OUTPUT_Q1`	Application (`App_OPC_UA_RES2`)	`logiBUS::io::DQ::logiBUS_QXA`	Schaltet `Output_Q1`, nimmt den Zustand über AX-Adapter-Socket entgegen
`CLIENT_Q1`	Resource `EMB_RES_A` (Gerät A)	`adapter::net::AX_CLIENT_1_0`	Aktiver Remote-Write auf den von Gerät B überwachten OPC-UA-Knoten
`SUBSCRIBE_Q1`	Resource `EMB_RES_B` (Gerät B)	`adapter::net::AX_SUBSCRIBE_1`	Lokal überwachter OPC-UA-Knoten, wird von Gerät A remote beschrieben

Die Application selbst enthält **keine** OPC-UA-Bausteine — sie ist bewusst so gehalten, wie sie auch in einer nicht-verteilten Übung aussehen würde (`INPUT_I1.IN` ⟶ `OUTPUT_Q1.OUT`, direkte Plug→Socket-Adapterverbindung, identisch zum Muster in `Uebung_001.SUB`). Erst das `Mapping` verteilt beide FBs auf

unterschiedliche Geräte; die eigentliche Netzwerk-Übertragung übernehmen `CLIENT_Q1`/`SUBSCRIBE_Q1` unten in den jeweiligen Resource-FBNetworks.

OPC-UA-Adressraum

Die Stationen `PC_A`/`PC_B` (`MGR_ID` der Geräte) stehen nur in `myOpcUaAddresses.gcf`; die OPC-UA-IDs berechnet die GCF mit den Functions der Lib `MyLib_OPC` (`F_OPC_LOCAL`, `F_OPC_REMOTE`, `F_MGR_TO_OPC`). Die Namen `*_LOCAL_*`/`*_REMOTE_*` bleiben dabei erhalten. Aufbau und Berechnung: OPC-UA-Adressen.

Beide Konstanten liegen in `VV::const::OPC_UA::myOpcUaAddresses` und teilen sich denselben Node-Pfad/Namen — nur ACTION und (bei `CLIENT`) der Ziel-Endpoint unterscheiden sich:

Konstante	Wert	Verwendet von
`Q1_LOCAL_READ`	`opc_ua[READ;/Objects/ DigitalOutput/Q1,1:s=Output_Q1]`	`SUBSCRIBE_Q1` auf Gerät B (lokal überwachter Knoten, `Local|READ| SUBSCRIBE`)
`Q1_REMOTE_WRITE`	`opc_ua[WRITE;opc.tcp://192.168.1.12:4840#1/ Objects/ DigitalOutput/Q1,1:s=Output_Q1]`	`CLIENT_Q1` auf Gerät A (aktiver Remote-Write, `Remote|WRITE|CLIENT`)

Programmablauf und Verbindungen

1. **Application** (`App_OPC_UA_RES2`): `INPUT_I1` (`logiBUS_IXA`, `Input_I1`) ist über eine Adapterverbindung (`INPUT_I1.IN` → `OUTPUT_Q1.OUT`) direkt mit `OUTPUT_Q1` (`logiBUS_QXA`, `Output_Q1`) verbunden — das rein logische Modell, unabhängig von der späteren Geräteverteilung.
2. **Mapping**: `INPUT_I1` → `FORTE_PC_A.EMB_RES_A`, `OUTPUT_Q1` → `FORTE_PC_B.EMB_RES_B`.
3. **Resource `EMB_RES_A`** (Gerät A): `START.COLD` initialisiert zuerst `App_OPC_UA_RES2.INPUT_I1` selbst (dotted-path-Referenz auf dessen `.INIT`); erst dessen `.INITO`-Bestätigung löst `CLIENT_Q1.INIT` aus. Die eigentliche Adapterverbindung `App_OPC_UA_RES2.INPUT_I1.IN` → `CLIENT_Q1.IN` überbrückt die Mapping-Grenze per dotted-path-Referenz — Application-Plug und Resource-Socket direkt verbunden, ohne Zwischenbaustein.
4. **Resource `EMB_RES_B`** (Gerät B): spiegelbildlich — `START.COLD` → `SUBSCRIBE_Q1.INIT`, dessen `.INITO` → `App_OPC_UA_RES2.OUTPUT_Q1.INIT`, und `SUBSCRIBE_Q1.OUT` → `App_OPC_UA_RES2.OUTPUT_Q1.OUT` (wieder direkte Plug→Socket-Adapterverbindung über die Mapping-Grenze hinweg).
5. **Laufzeit**: Ändert sich `Input_I1` auf Gerät A, schreibt `CLIENT_Q1` den neuen Wert per OPC-UA auf `opc.tcp://192.168.1.12:4840` in den von `SUBSCRIBE_Q1` überwachten Knoten — Gerät B übernimmt den Wert und schaltet `Output_Q1`. Diese Verbindung existiert **nicht** als Modell- Verbindung in der `.sys`-Datei, sondern ausschließlich als OPC-UA-Netzwerkkommunikation zur Laufzeit.

Technische Besonderheiten

- **Adapterverbindungen können die Mapping-Grenze überschreiten**: Anders als ursprünglich angenommen, akzeptiert 4diac eine `AdapterConnections`- Verbindung zwischen einem Application-FB-Pin und einem Resource-FB-Pin per dotted-path-Referenz — genau wie bei einer reinen Event-/Datenverbindung (siehe `Training_01_PUBLISH_SUBSCRIBE`). Ein separater Konvertierungs- baustein (`AX_BOOL_TO_X`/`AX_X_TO_BOOL`) ist deshalb hier nicht nötig, weil `INPUT_I1`/`OUTPUT_Q1` selbst bereits adapterbasiert sind (`logiBUS_IXA`/`logiBUS_QXA`).
- **Zweistufige Initialisierung**: `START.COLD` initialisiert zuerst den physischen I/O-Baustein (`INPUT_I1`/`OUTPUT_Q1`), erst dessen `INITO` triggert den Kommunikationsbaustein (`CLIENT_Q1`/`SUBSCRIBE_Q1`) — so ist sichergestellt, dass die I/O-Anbindung steht, bevor der OPC-UA-Kanal aktiv wird.

Lernziele

- Kanonisches IEC-61499-Verteilungsmuster: Application bleibt geräte- neutrale, portable Logik; die Resource jedes Geräts trägt die protokollspezifische Kommunikation.
- Korrekte ACTION-Zuordnung für lokal überwachte (`READ`/`SUBSCRIBE`) und remote geschriebene (`WRITE`/`CLIENT`) OPC-UA-Knoten.
- Explizite Initialisierungsreihenfolge (`START.COLD` → I/O-Baustein → Kommunikationsbaustein) statt blindem Verlassen auf automatisches EInit-Feuern.

Schwierigkeitsgrad: Mittel **Vorkenntnisse**: `Uebung_001.SUB` (Grundmuster I1→Q1), Grundlagen der OPC-UA-Adapterbausteine (`AX_CLIENT_1_0`/`AX_SUBSCRIBE_1`), IEC-61499- Mapping/Distribution-Konzepte.

Zusammenfassung

`Training_02_OPC_UA_RES` überträgt das triviale `Uebung_001.SUB`-Muster (`Input_I1` ⟶ `Output_Q1`) auf zwei physisch getrennte FORTE-Geräte und demonstriert dabei das kanonische IEC-61499-„RES style“-Verteilungsmuster: portable Logik auf Application-Ebene, Kommunikation auf Resource-Ebene, per dotted-path-Adapterverbindung über die Mapping-Grenze hinweg verknüpft.

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

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

2.3.3.4 Training 03 OPC UA RES

Training_03_OPC_UA_RES: Toggle-Flipflop über OPC-UA („SUB style“)



• – ◦ • – ◦ • – ◦ •

Einleitung

`Training_03_OPC_UA_RES` überträgt `Uebung_010d_AX` (SoftKey-getriggertes Toggle-Flipflop mit `GreenWhiteBackground`, siehe `Uebung_010d_AX.md`) auf zwei physisch getrennte FORTE-Geräte, verbunden über OPC-UA.

Anders als `Training_02_OPC_UA_RES` („RES style“: die Kommunikationsbausteine stecken in der `<Resource>` jedes Geräts) verwendet diese Übung den **„SUB style“**: das gesamte Protokoll — SoftKey-Eingabe, OPC-UA-Client/ -Server-Bausteine und `GreenWhiteBackground` — steckt in zwei wiederverwendbaren `MyLib::sys`-Composites (`Uebung_010d_PC_A_OPC`, `Uebung_010d_PC_B_OPC`). Die `.sys`-Datei selbst enthält dadurch **keine** FBs mehr, nur noch zwei `SubApp`-Instanzen und das Geräte-`Mapping`.

Verwendete Composites

Composite	Gerät	Zweck
`Uebung_010d_PC_A_OPC`	A (Station 11, `192.168.1.11`)	`SoftKey_F1` löst per `CLIENT_0` einen argument- und rückgabewertlosen OPC-UA-Methodenaufruf auf Gerät B aus; `GreenWhiteBackground` zeigt den von Gerät B lokal überwachten Zustand
`Uebung_010d_PC_B_OPC`	B (Station 12, `192.168.1.12`)	Empfängt den Methodenaufruf per `SERVER_0`, taktet `AX_T_FF` (Toggle-Flipflop), schaltet `Output_Q1` und schreibt den neuen Zustand aktiv per `AX_CLIENT_1_0` zurück auf Gerät A

Auf Gerät A ist **kein** `AX_T_FF` nötig — der SoftKey-Druck ist ein reiner RPC-Trigger ohne Wertwechsel-Trick. Auf Gerät A liest ein `AX_SUBSCRIBE_1` (`STATE_SUBSCRIBE`) den von Gerät B geschriebenen Zustand lokal und speist `GreenWhiteBackground_AX`.

OPC-UA-Adressraum

Die Stationen `PC_A`/`PC_B` (`MGR_ID` der Geräte) stehen nur in `myOpcUaAddresses.gcf`; die OPC-UA-IDs berechnet die GCF mit den Functions der Lib `MyLib_OPC` (`F_OPC_LOCAL`, `F_OPC_REMOTE`, `F_MGR_TO_OPC`). Die Namen `*_LOCAL_*`/`*_REMOTE_*` bleiben dabei erhalten. Aufbau und Berechnung: OPC-UA-Adressen.

Beide Konstanten-Paare liegen in `VV::const::OPC_UA::myOpcUaAddresses`:

Konstante	Wert	Verwendet von
`FF1_TRIGGER_LOCAL_METHOD`	`opc_ua[CREATE_METHOD;/ Objects/ ToggleFF/FF1_Trigger]`	`TRIGGER_SERVER` auf Gerät B (lokal erzeugte Methode)
`FF1_TRIGGER_REMOTE_CALL`	`opc_ua[CALL_METHOD;opc.tcp://192.168.1.12:4840/ Objects/ ToggleFF/FF1_Trigger]`	`TRIGGER_CLIENT` auf Gerät A (ruft dieselbe Methode remote auf)
`FF1_STATE_LOCAL_READ`	`opc_ua[READ;/Objects/ ToggleFF/ FF1_State,1:s=FF1_State]`	`STATE_SUBSCRIBE` auf Gerät A (lokal überwachter Zustandsknoten)
`FF1_STATE_REMOTE_WRITE`	`opc_ua[WRITE;opc.tcp://192.168.1.11:4840/ Objects/ToggleFF/ FF1_State,1:s=FF1_State]`	`STATE_CLIENT` auf Gerät B (schreibt denselben Knoten remote)

Anders als `Q1_LOCAL_READ`/`Q1_REMOTE_WRITE` in Training_02 (reiner Werte- Transport) sind `FF1_TRIGGER_*` eine echte OPC-UA-**Methode** (RPC), nicht ein geschriebener/gelesener Knoten — passend zum ereignisgetriebenen SoftKey-Tastendruck, der keinen Wert trägt.

Programmablauf und Verbindungen

1. **`Uebung_010d_PC_A_OPC`** (Gerät A): `SoftKey_UP_F1.IND` (Softkey losgelassen) löst `TRIGGER_CLIENT.REQ` aus — `CLIENT_0` ruft die Remote-Methode `FF1_TRIGGER_REMOTE_CALL` auf. Parallel liefert `STATE_SUBSCRIBE` (`AX_SUBSCRIBE_1`) den von Gerät B geschriebenen Zustand per Adapterverbindung an `GreenWhiteBackground_AX.DI1`.
2. **`Uebung_010d_PC_B_OPC`** (Gerät B): `TRIGGER_SERVER.IND` (Methode aufgerufen) taktet `AX_T_FF.CLK` — das Toggle-Flipflop invertiert seinen Zustand. `AX_SPLIT_2` verteilt den neuen Zustand an zwei Ziele: `DigitalOutput_Q1.OUT` (lokal, schaltet `Output_Q1`) und `STATE_CLIENT.IN` (`AX_CLIENT_1_0`, schreibt den Zustand aktiv nach `FF1_STATE_REMOTE_WRITE` zurück auf Gerät A).
3. **Mapping/Devices**: `App_OPC_UA_RES3.SubApp_PC_A` → `FORTE_PC_A`, `SubApp_PC_B` → `FORTE_PC_B`; je ein `E_TRIG('EInit')` pro Gerät initialisiert das jeweilige Composite beim Start.

Technische Besonderheiten

- **RPC statt Wertübertragung**: `FF1_TRIGGER_*` nutzt `CALL_METHOD`/ `CREATE_METHOD` statt `READ`/`WRITE` — der richtige ACTION-Typ für einen argumentlosen, zustandslosen Auslöser (ein Tastendruck hat keinen Wert, der übertragen werden müsste).
- **„SUB style“ vs. „RES style“**: Das Protokoll steckt hier vollständig im wiederverwendbaren `MyLib::sys`-Composite, nicht in der Resource des Geräts. Der Vorteil: dasselbe Composite lässt sich unverändert auf jedem Gerät instanzieren, das dieselbe Rolle (A oder B) spielt — die `.sys`- Datei bleibt auf Mapping und Geräte-Setup reduziert.
- **Kein Wertwechsel-Trick nötig**: Da der Trigger eine echte RPC-Methode ist (nicht ein beobachteter Wert), braucht Gerät A kein eigenes `AX_T_FF` — anders als bei reiner Read/Write-basierter Signalisierung, wo ein Flankenwechsel simuliert werden müsste.

Lernziele

- OPC-UA-Methodenaufrufe (`CALL_METHOD`/`CREATE_METHOD`) als RPC-Alternative zu `READ`/`WRITE` für ereignisgetriggerte, wertlose Trigger.
- „SUB style“-Verteilungsmuster: Protokoll im wiederverwendbaren Composite statt in der Resource.
- Rückweg (Zustand von B nach A) über einen zweiten, unabhängigen Read/Write-Adresskanal, getrennt vom Trigger-Kanal.

Schwierigkeitsgrad: Mittel **Vorkenntnisse**: `Uebung_010d_AX.SUB` (lokales Toggle-Flipflop-Muster), `Training_02_OPC_UA_RES` (RES-style-Verteilung, `AX_CLIENT_1_0`/ `AX_SUBSCRIBE_1`).

Zusammenfassung

`Training_03_OPC_UA_RES` verteilt `Uebung_010d_AX` auf zwei Geräte und führt dabei zwei neue Konzepte gegenüber Training_02 ein: OPC-UA-Methodenaufrufe als RPC-Alternative zur reinen Wertübertragung, und das „SUB style"-Muster, bei dem das Protokoll im wiederverwendbaren Composite statt in der Resource steckt.

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

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

2.3.3.5 Training 04 OPC UA RES

Training_04_OPC_UA_RES: SR+Toggle-Flipflop über 3 OPC-UA-Methoden („SUB style", Option A)



• – ◦ • – ◦ • – ◦ •

Einleitung

`Training_04_OPC_UA_RES` überträgt `Uebung_010e_AX` (SR+Toggle-Flipflop über 3 SoftKeys, siehe `Uebung_010e_AX.md`) nach demselben „SUB style"-Muster wie `Training_03_OPC_UA_RES` auf zwei Geräte. Set, Reset und Toggle sind drei **eigenständige** OPC-UA-Methoden (**Option A**) — Gerät A ruft je nach gedrücktem SoftKey eine von drei `CLIENT_0`-Instanzen auf, statt eine einzelne Methode mit einem String-Parameter zu übergeben.

Verwendete Composites

Composite	Gerät	Zweck
`Uebung_010e_PC_A_OPC`	A (`192.168.1.11`)	3 SoftKeys (Set/Reset/Toggle) rufen je einen eigenen `CLIENT_0`-Methodenaufruf auf Gerät B auf; `GreenWhiteBackground` am Toggle-SoftKey zeigt den überwachten Zustand
`Uebung_010e_PC_B_OPC`	B (`192.168.1.12`)	3 `SERVER_0`-Instanzen empfangen Set/Reset/Toggle, treiben `AX_T_FF_SR`, schalten `Output_Q1` und schreiben den Zustand per `AX_CLIENT_1_0` zurück

OPC-UA-Adressraum

Die Stationen `PC_A`/`PC_B` (`MGR_ID` der Geräte) stehen nur in `myOpcUaAddresses.gcf`; die OPC-UA-IDs berechnet die GCF mit den Functions der Lib `MyLib_OPC` (`F_OPC_LOCAL`, `F_OPC_REMOTE`, `F_MGR_TO_OPC`). Die Namen `*_LOCAL_*`/`*_REMOTE_*` bleiben dabei erhalten. Aufbau und Berechnung: OPC-UA-Adressen.

Drei unabhängige Methoden-Adresspaare plus ein Zustands-Adresspaar, alle in `VV::const::OPC_UA::myOpcUaAddresses`:

Konstante (lokal/remote)	Wert (Auszug)	Zweck
`SR1_SET_LOCAL_METHOD` / `SR1_SET_REMOTE_CALL`	`.../SRFlipflop/SR1_Set`	Set-Methode
`SR1_RESET_LOCAL_METHOD` / `SR1_RESET_REMOTE_CALL`	`.../SRFlipflop/SR1_Reset`	Reset-Methode
`SR1_TOGGLE_LOCAL_METHOD` / `SR1_TOGGLE_REMOTE_CALL`	`.../SRFlipflop/SR1_Toggle`	Toggle-Methode
`SR1_STATE_LOCAL_READ` / `SR1_STATE_REMOTE_WRITE`	`.../SRFlipflop/SR1_State`	Zustands-Rückkanal (`READ`/`WRITE`, wie in Training_03)

Jede der drei Methoden hat einen eigenen, unabhängigen `CLIENT_0`/`SERVER_0`- Adresspfad (`SR1_Set`, `SR1_Reset`, `SR1_Toggle`) — keine gemeinsam genutzte Methode mit Parameter.

Programmablauf und Verbindungen

1. **`Uebung_010e_PC_A_OPC`** (Gerät A): `SoftKey_SET.IND` → `TRIGGER_SET_CLIENT.REQ`, `SoftKey_RESET.IND` → `TRIGGER_RESET_CLIENT.REQ`, `SoftKey_TOGGLE.IND` → `TRIGGER_TOGGLE_CLIENT.REQ` — drei unabhängige `CLIENT_0`-Aufrufe, je einer pro SoftKey. `STATE_SUBSCRIBE` speist wie in Training_03 `GreenWhiteBackground_AX.DI1` am Toggle-SoftKey.
2. **`Uebung_010e_PC_B_OPC`** (Gerät B): drei `SERVER_0`-Instanzen (`TRIGGER_SET_SERVER`, `TRIGGER_RESET_SERVER`, `TRIGGER_TOGGLE_SERVER`) takten je den passenden Eingang von `AX_T_FF_SR` (`S`/`R`/`CLK`); der neue Zustand geht wie in Training_03 über `AX_SPLIT_2` an `Output_Q1` und `STATE_CLIENT` (Rückschreiben nach Gerät A).

Technische Besonderheiten

- **Option A: eigene Methode pro Aktion** — statt einer gemeinsamen Methode mit einem Parameter (der signalisieren würde „welche Aktion“) bekommt jede Aktion ihren eigenen `CLIENT_0`/`SERVER_0`-Adresspfad. Das ist einfacher zu verdrahten (keine Parameterauswertung nötig), verbraucht aber drei statt einer Server-Instanz.
- **Set/Reset/Toggle auf einem Flipflop**: `AX_T_FF_SR` ist die SR+Toggle-fähige Variante von `AX_T_FF` (siehe `Uebung_010e_AX.md` für die lokale ECC-Logik).

Lernziele

- Mehrere unabhängige OPC-UA-Methoden auf demselben Gerät, je mit eigenem Adresspfad statt gemeinsamem Parameter.
- SR+Toggle-Flipflop-Logik (`AX_T_FF_SR`) ferngesteuert über drei getrennte RPC-Kanäle.

Schwierigkeitsgrad: Mittel bis fortgeschritten **Vorkenntnisse**: `Uebung_010e_AX.SUB`, `Training_03_OPC_UA_RES` (RPC-Muster, „SUB style“).

Zusammenfassung

`Training_04_OPC_UA_RES` erweitert das RPC-Muster aus Training_03 auf drei unabhängige Aktionen (Set/Reset/Toggle), jede mit eigenem OPC-UA-Methoden- Adresspfad („Option A“) statt eines parametrisierten Einzelaufrufs.

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

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

2.3.3.6 Training 05 OPC UA RES

Training_05_OPC_UA_RES: reines SR-Flipflop über 2 OPC-UA-Methoden („SUB style“)



• – ◦ • – ◦ • – ◦ •

Einleitung

`Training_05_OPC_UA_RES` ist die Set/Reset-**ohne**-Toggle-Variante von `Training_04_OPC_UA_RES`: 2 SoftKeys (Set, Reset) statt 3, ein reines `AX_SR`-Flipflop statt `AX_T_FF_SR`. Gleiches „SUB style“-Muster, gleiche Option-A-Methodenaufteilung (eigene Methode pro Aktion), aber ohne die Toggle-Aktion und ohne den dritten SoftKey/Server.

Verwendete Composites

Composite	Gerät	Zweck
`SoftKeySR_PC_A_OPC`	A (`192.168.1.11`)	2 SoftKeys (Set/Reset) rufen je einen eigenen `CLIENT_0`-Methodenaufruf auf Gerät B auf; `GreenWhiteBackground` am Set-SoftKey zeigt den überwachten Zustand
`SoftKeySR_PC_B_OPC`	B (`192.168.1.12`)	2 `SERVER_0`-Instanzen empfangen Set/Reset, treiben `AX_SR`, schalten `Output_Q1` und schreiben den Zustand per `AX_CLIENT_1_0` zurück

OPC-UA-Adressraum

Die Stationen `PC_A`/`PC_B` (`MGR_ID` der Geräte) stehen nur in `myOpcUaAddresses.gcf`; die OPC-UA-IDs berechnet die GCF mit den Functions der Lib `MyLib_OPC` (`F_OPC_LOCAL`, `F_OPC_REMOTE`, `F_MGR_TO_OPC`). Die Namen `*_LOCAL_*`/`*_REMOTE_*` bleiben dabei erhalten. Aufbau und Berechnung: OPC-UA-Adressen.

Konstante (lokal/remote)	Wert (Auszug)	Zweck
`SR2_SET_LOCAL_METHOD` / `SR2_SET_REMOTE_CALL`	`.../SRFlipflop/SR2_Set`	Set-Methode
`SR2_RESET_LOCAL_METHOD` / `SR2_RESET_REMOTE_CALL`	`.../SRFlipflop/SR2_Reset`	Reset-Methode
`SR2_STATE_LOCAL_READ` / `SR2_STATE_REMOTE_WRITE`	`.../SRFlipflop/SR2_State`	Zustands-Rückkanal

Eigener Adress-Namensraum (`SR2_*`) getrennt von `SR1_*` aus Training_04 — beide Übungen können unverändert gleichzeitig auf denselben zwei Geräten laufen, ohne OPC-UA-Adresskollision.

Programmablauf und Verbindungen

1. **SoftKeySR_PC_A_OPC** (Gerät A): SoftKey_SET.IND ⟶ TRIGGER_SET_CLIENT.REQ, SoftKey_RESET.IND ⟶ TRIGGER_RESET_CLIENT.REQ — zwei unabhängige CLIENT_0-Aufrufe. STATE_SUBSCRIBE speist GreenWhiteBackground_AX.DI1 am Set-SoftKey.
2. **SoftKeySR_PC_B_OPC** (Gerät B): TRIGGER_SET_SERVER.IND ⟶ AX_SR.S, TRIGGER_RESET_SERVER.IND ⟶ AX_SR.R; der neue Zustand geht wie in Training_04 über AX_SPLIT_2 an Output_Q1 und STATE_CLIENT (Rückschreiben nach Gerät A).

Technische Besonderheiten

- **Reduzierte Aktionsmenge**: AX_SR hat nur S/R/Q — kein CLK für Toggle. Wer nur Set/Reset braucht, spart sich Composite-Komplexität und eine dritte Methode/Server-Instanz gegenüber Training_04.
- **Gleiches Grundmuster wie Training_04**: Bis auf die fehlende Toggle-Aktion sind Verdrahtung, Rückschreibe-Mechanismus und Adressstruktur identisch — eine gute Vorlage, um Option A auf beliebig viele/wenige unabhängige Aktionen zu skalieren.

Lernziele

- Reines SR-Flipflop (AX_SR) ferngesteuert über zwei getrennte OPC-UA-Methoden, als reduzierte Variante von Training_04.
- Eigenständiger OPC-UA-Adress-Namensraum pro Übung, um Kollisionen bei gleichzeitigem Betrieb mehrerer Trainings-Systeme zu vermeiden.

Schwierigkeitsgrad: Mittel **Vorkenntnisse**: Training_04_OPC_UA_RES (gleiches Muster mit Toggle).

Zusammenfassung

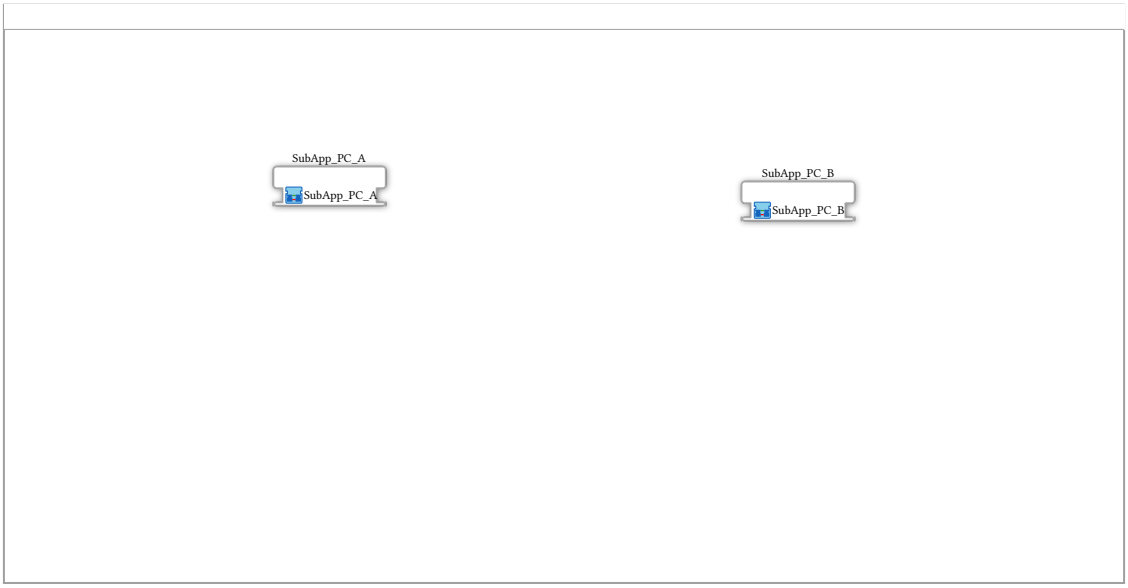
Training_05_OPC_UA_RES reduziert Training_04 auf reines Set/Reset ohne Toggle — gleiches „SUB style“/ Option-A-Muster, zwei statt drei OPC-UA- Methoden, eigener Adress-Namensraum.

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

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

2.3.3.7 Training 06 OPC UA RES

Training_06_OPC_UA_RES: SR+Toggle über OPC-UA — Adapter-Variante von Training_04



• – ◦ • – ◦ • – ◦ •

Einleitung

Training_06_OPC_UA_RES ist die **Adapter-Variante** von Training_04_OPC_UA_RES — dieselbe Funktion (3 SoftKeys, SR+Toggle- Flipflop, OPC-UA-Rückschreiben), aber statt einzelner CLIENT_0/SERVER_0/ AX_CLIENT_1_0-Instanzen mit manueller Datenverdrahtung kommen die kombinierten Adapter-Baustei-

ne aus der adapter-3.0.0-Bibliothek zum Einsatz (siehe die adapter-3.0.0-FB-Familie, dokumentiert unter Bibliotheken/ExternalLibraries/adapter). Gleiche OPC-UA-Adressen wie Training_04 (SR1_*).

Verwendete Composites

Composite	Gerät	Zweck
Uebung_010e_PC_A_OPC_Adapter	A (192.168.1.11)	HMI (3 SoftKeys + GreenWhiteBackground) hinter SoftKeySRT_ASRT_AX, OPC-UA-Trigger hinter ASRT_AX_CLIENT_0_SUBSCRIBE_1 — beide über **eine** bidirektionale Adapterverbindung gekoppelt
Uebung_010e_PC_B_OPC_Adapter	B (192.168.1.12)	ASRT_AX_SERVER_0_CLIENT_1_0 bündelt die 3 SERVER_0 + AX_CLIENT_1_0 hinter einem Adapter; ASRT_AX_AX_SPLIT verteilt an DigitalOutput_Q1 und ASRT_AX_T_FF_SR_2

OPC-UA-Adressraum

Identisch zu Training_04 (SR1_SET_*, SR1_RESET_*, SR1_TOGGLE_*, SR1_STATE_*) — siehe Training_04_OPC_UA_RES. Nur die Verdrahtung im Composite ändert sich, nicht der Adressraum.

Programmablauf und Verbindungen

1. **Uebung_010e_PC_A_OPC_Adapter** (Gerät A): SoftKeySRT_ASRT_AX bündelt die 3 SoftKeys + GreenWhiteBackground hinter einem ASRT_AX-Plug; eine einzige Adapterverbindung (SoftKeySRT_ASRT_AX.OUT → TRIGGER.S_R_T) übergibt Set/Reset/Toggle/ Zustand an ASRT_AX_CLIENT_0_SUBSCRIBE_1 (TRIGGER), das die 3 ID_*_CALL- und ID_STATE_READ-Adressen als Datenpins trägt.
2. **Uebung_010e_PC_B_OPC_Adapter** (Gerät B): TRIGGER (ASRT_AX_SERVER_0_CLIENT_1_0) trägt die 4 ID_*-Adressen als Datenpins und liefert das gebündelte Set/Reset/Toggle/Zustand-Signal über TRIGGER.S_R_T an SPLIT (ASRT_AX_AX_SPLIT), das es an FLIPFLOP (ASRT_AX_T_FF_SR_2) weiterreicht; SPLIT.AX_OUT schaltet parallel DigitalOutput_Q1.

Technische Besonderheiten

- **Ein Adapter statt vier Einzelbausteine**: ASRT_AX_SERVER_0_CLIENT_1_0 bündelt 3× SERVER_0 + 1× AX_CLIENT_1_0 hinter einem einzigen bidirektionalen Adapter-Anschluss — die vier ID_*-Adresspins bleiben als Datenpins direkt am Composite-Interface sichtbar (siehe PR #158 für die vollständige adapter-3.0.0-FB-Dokumentation dieser Kombi-Typen).
- **Durchgängige Adapterkette statt Event/BOOL-Wechsel**: Anders als in Training_04 (AX_T_FF_SR mit separaten S/R/CLK-Eventpins und BOOL-Q) läuft hier das gesamte Set/Reset/Toggle/Zustand-Bündel als **eine** Adapterverbindung durch TRIGGER → SPLIT → FLIPFLOP — die eigentliche Flipflop-Logik (ASRT_AX_T_FF_SR_2) ist selbst adapterbasiert, nicht mehr auf Einzel-Events verdrahtet.
- **Weniger Verdrahtungsaufwand, gleiche Semantik**: Funktional identisch zu Training_04, aber deutlich weniger Verbindungen im Composite — der Preis ist, dass die einzelnen Aktionen (Set/Reset/Toggle) nicht mehr als separate Pins im Netzplan sichtbar sind, sondern im Adapterbündel verschwinden (nur noch an den ID_*-Datenpins ablesbar).

Lernziele

- Adapter-3.0.0-Kombi-Bausteine (ASRT_AX_CLIENT_0_SUBSCRIBE_1, ASRT_AX_SERVER_0_CLIENT_1_0) als Ersatz für mehrere einzeln verdrahtete Netzwerk-FBs.
- Durchgängige bidirektionale Adapterketten (ASRT_AX_AX_SPLIT, ASRT_AX_T_FF_SR_2) statt Einzel-Event-/BOOL-Verdrahtung.
- Direkter Vergleich Training_04 (manuell) vs. Training_06 (Adapter) bei identischer Funktion und identischem OPC-UA-Adressraum.

Schwierigkeitsgrad: Fortgeschritten **Vorkenntnisse**: Training_04_OPC_UA_RES, adapter-3.0.0-FB-Familie (Bibliotheken/ExternalLibraries/adapter).

Zusammenfassung

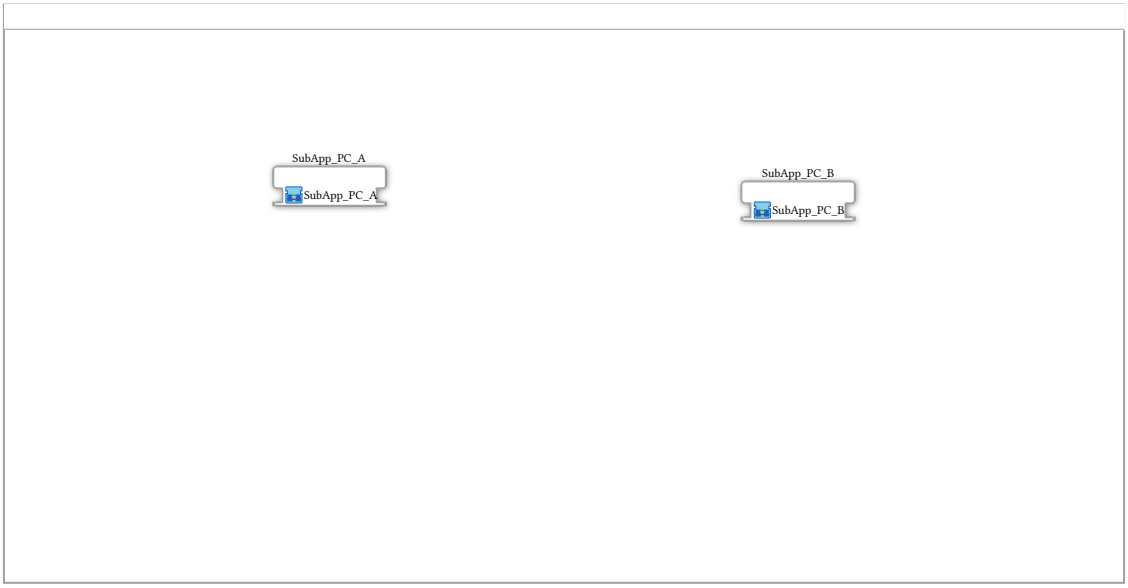
`Training_06_OPC_UA_RES` zeigt dieselbe SR+Toggle-über-OPC-UA-Funktion wie Training_04, jetzt mit den kombinierten adapter-3.0.0-Bausteinen (`ASRT_AX_CLIENT_0_SUBSCRIBE_1`, `ASRT_AX_SERVER_0_CLIENT_1_0`) statt manuell verdrahteter Einzel-FBs — weniger Verbindungen, gleicher Adressraum, gleiches Verhalten.

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

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

2.3.3.8 Training 07 OPC UA RES

Training_07_OPC_UA_RES: reines SR-Flipflop über OPC-UA — Adapter-Variante von Training_05



• – ◦ • – ◦ • – ◦ •

Einleitung

`Training_07_OPC_UA_RES` verhält sich zu `Training_05_OPC_UA_RES` genauso wie `Training_06_OPC_UA_RES` zu `Training_04_OPC_UA_RES`: dieselbe Set/Reset-Funktion, aber mit den kombinierten `ASR_AX`-Adapterbausteinen statt einzeln verdrahteter `CLIENT_0/SERVER_0/AX_CLIENT_1_0`-Instanzen. Gleiche OPC-UA-Adressen wie Training_05 (`SR2_*`).

Verwendete Composites

Composite	Gerät	Zweck
`SoftKeySR_PC_A_OPC_Adapter`	A (`192.168.1.11`)	HMI (2 SoftKeys + `GreenWhiteBackground`) hinter `SoftKeySR_ASR_AX`, OPC-UA-Trigger hinter `ASR_AX_CLIENT_0_SUBSCRIBE_1` — über eine bidirektionale Adapterverbindung gekoppelt
`SoftKeySR_PC_B_OPC_Adapter`	B (`192.168.1.12`)	`ASR_AX_SERVER_0_CLIENT_1_0` bündelt die 2 `SERVER_0` + `AX_CLIENT_1_0` hinter einem Adapter; `ASR_AX_AX_SPLIT` verteilt an `DigitalOutput_Q1` und `ASR_AX_SR_2`

OPC-UA-Adressraum

Identisch zu Training_05 (`SR2_SET_*`, `SR2_RESET_*`, `SR2_STATE_*`) — siehe Training_05_OPC_UA_RES.

Programmablauf und Verbindungen

1. **`SoftKeySR_PC_A_OPC_Adapter`** (Gerät A): `SoftKeySR_ASR_AX` bündelt die 2 SoftKeys + `GreenWhiteBackground` hinter einem `ASR_AX`-Plug; eine Adapterverbindung übergibt Set/Reset/Zustand an `ASR_AX_CLIENT_0_SUBSCRIBE_1` (`TRIGGER`), das die `ID_SET_CALL`/ `ID_RESET_CALL`/ `ID_STATE_READ`-Adressen trägt.
2. **`SoftKeySR_PC_B_OPC_Adapter`** (Gerät B): `TRIGGER` (`ASR_AX_SERVER_0_CLIENT_1_0`) trägt die 3 `ID_*`-Adressen und liefert das Set/Reset/Zustand-Bündel über `TRIGGER.S_R` an `SPLIT` (`ASR_AX_AX_SPLIT`), das es an `FLIPFLOP` (`ASR_AX_SR_2`) weiterreicht; `SPLIT.AX_OUT` schaltet parallel `DigitalOutput_Q1`.

Technische Besonderheiten

- **`ASR_AX_*`-Kombis statt `ASRT_AX_*`**: Dieselbe Bündelungslogik wie in Training_06, aber die zweiwertige (Set/Reset) statt dreiwertige (Set/Reset/Toggle) Adapter-Variante — ein Pin (`S_R` statt `S_R_T`) weniger, sonst strukturell identisch.
- **Direkter Vergleich mit Training_05**: Gleicher Adressraum, gleiche Funktion, aber deutlich weniger Einzelverbindungen im Composite.

Lernziele

- Adapter-3.0.0-Kombi-Bausteine für die reduzierte Set/Reset-ohne-Toggle- Variante (`ASR_AX_CLIENT_0_SUBSCRIBE_1`, `ASR_AX_SERVER_0_CLIENT_1_0`).
- Direkter Vergleich Training_05 (manuell) vs. Training_07 (Adapter) bei identischer Funktion.

Schwierigkeitsgrad: Fortgeschritten **Vorkenntnisse**: `Training_05_OPC_UA_RES`, `Training_06_OPC_UA_RES` (gleiches Adapter-Muster mit Toggle).

Zusammenfassung

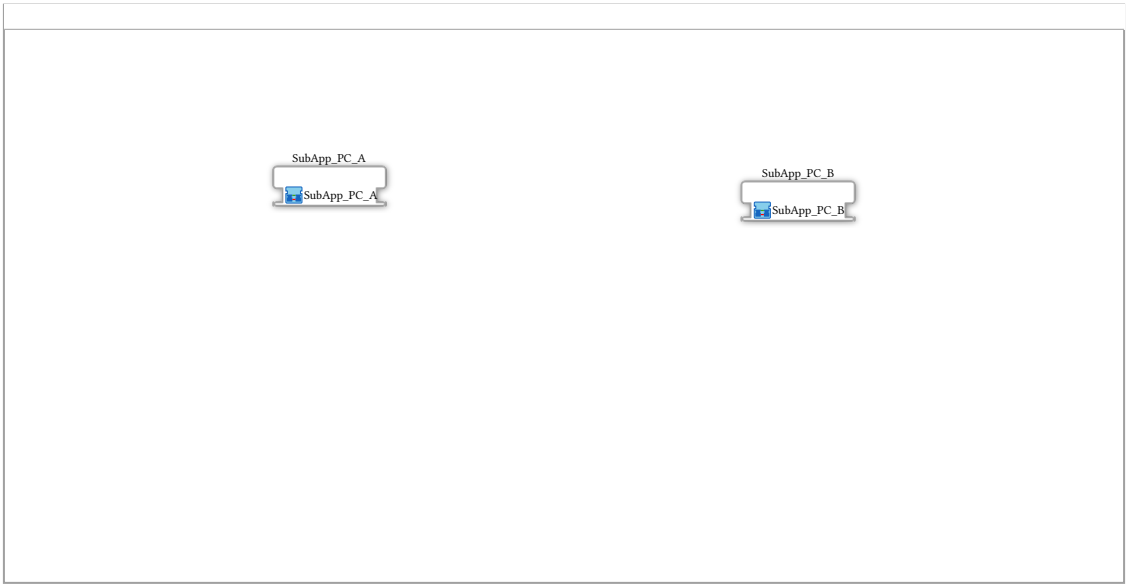
`Training_07_OPC_UA_RES` überträgt das Adapter-Bündelungsmuster aus Training_06 auf die reduzierte Set/Reset-Variante aus Training_05.

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

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

2.3.3.9 Training 08 OPC UA RES

Training_08_OPC_UA_RES: Toggle-Flipflop über OPC-UA — Vergleichsbasis für Training_09



• – ◦ • – ◦ • – ◦ •

Einleitung

`Training_08_OPC_UA_RES` verwendet exakt dieselben Composites wie `Training_03_OPC_UA_RES` (`Uebung_010d_PC_A_OPC`/`Uebung_010d_PC_B_OPC`) und dieselben OPC-UA-Adressen (`FF1_*`) — funktional identisch zu Training_03.

Der Grund für die eigenständige `.sys`-Datei: Training_08 dient als direkte **Vergleichsbasis (manuelle Verdrahtung)** unmittelbar neben `Training_09_OPC_UA_RES` (Adapter-Variante), analog zum Paar Training_04/Training_06 und Training_05/Training_07. Ohne Training_08 müsste man für den Vorher/Nachher-Vergleich der Adapter-Bündelung bei der Toggle-only-Variante wieder auf Training_03 zurückgreifen, das an anderer Stelle in der Serie steht.

Verwendete Composites

Identisch zu Training_03 — siehe Training_03_OPC_UA_RES für die vollständige Beschreibung von `Uebung_010d_PC_A_OPC` und `Uebung_010d_PC_B_OPC`.

Composite	Gerät
`Uebung_010d_PC_A_OPC`	A (`192.168.1.11`)
`Uebung_010d_PC_B_OPC`	B (`192.168.1.12`)

OPC-UA-Adressraum

Identisch zu Training_03 (`FF1_TRIGGER_*`, `FF1_STATE_*`) — siehe Training_03_OPC_UA_RES.

Programmablauf und Verbindungen

Identisch zu Training_03 — Ablauf und Verbindungen sind dort im Detail beschrieben.

Technische Besonderheiten

- **Bewusste Wiederverwendung, keine neue Logik**: Training_08 führt keine neue Funktion ein. Es stellt sicher, dass die „Vorher"-Seite des Toggle-only-Adapter-Vergleichs (siehe `Training_09_OPC_UA_RES`) direkt neben ihrer „Nachher"-Seite in der Trainingsserie steht — genau wie Training_04/06 und Training_05/07 bereits als Paare nebeneinander stehen.

Lernziele

- Direkter, unmittelbar benachbarter Vorher/Nachher-Vergleich (manuell vs. Adapter) für alle drei Aktionsmengen der Serie (Toggle-only, SR-only, SR+Toggle).

Schwierigkeitsgrad: Mittel **Vorkenntnisse**: `Training_03_OPC_UA_RES` (identisches Muster).

Zusammenfassung

`Training_08_OPC_UA_RES` ist funktional identisch zu Training_03 und dient ausschließlich als unmittelbare Vergleichsbasis für die Adapter-Variante `Training_09_OPC_UA_RES`.

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

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

2.3.3.10 Training 09 OPC UA RES

Training_09_OPC_UA_RES: Toggle-Flipflop über OPC-UA — Adapter-Variante von Training_08



• – ◦ • – ◦ • – ◦ •

Einleitung

`Training_09_OPC_UA_RES` ist die Adapter-Variante von `Training_08_OPC_UA_RES` (und damit auch von `Training_03_OPC_UA_RES`) — komplettiert die Serie der drei Vorher/Nachher-Adapter-Paare:

Aktionsmenge	Manuell	Adapter
Toggle-only	Training_03 / **Training_08**	**Training_09**
Set/Reset	Training_05	Training_07
Set/Reset/Toggle	Training_04	Training_06

Statt der `ASRT_AX_*`/`ASR_AX_*`-Kombis aus Training_06/07 kommen hier die **einwertigen** `AE_AX_*`-Adapterbausteine zum Einsatz (`AE` = Adapter- Event, ein einzelner Trigger ohne Set/Reset-Unterscheidung). Gleiche OPC-UA-Adressen wie Training_03/08 (`FF1_*`).

Verwendete Composites

Composite	Gerät	Zweck
`SoftKeyT_PC_A_OPC_Adapter`	A (`192.168.1.11`)	HMI (SoftKey + `GreenWhiteBackground`) hinter `SoftKeyT_AE_AX`, OPC-UA-Trigger hinter `AE_AX_CLIENT_0_SUBSCRIBE_1`
`SoftKeyT_PC_B_OPC_Adapter`	B (`192.168.1.12`)	`AE_AX_SERVER_0_CLIENT_1_0` bündelt `SERVER_0` + `AX_CLIENT_1_0`; `AE_AX_AX_SPLIT` verteilt an `DigitalOutput_Q1` und `AE_AX_T_FF`

OPC-UA-Adressraum

Identisch zu Training_03/08 (`FF1_TRIGGER_*`, `FF1_STATE_*`) — siehe Training_03_OPC_UA_RES.

Programmablauf und Verbindungen

1. **`SoftKeyT_PC_A_OPC_Adapter`** (Gerät A): `SoftKeyT_AE_AX` bündelt SoftKey + `GreenWhiteBackground` hinter einem `AE_AX`-Plug; eine Adapterverbindung (`SoftKeyT_AE_AX.OUT` ⟶ `TRIGGER.TRIGGER`) übergibt Trigger/Zustand an `AE_AX_CLIENT_0_SUBSCRIBE_1` (`TRIGGER`).

2. **`SoftKeyT_PC_B_OPC_Adapter`** (Gerät B): `TRIGGER` (`AE_AX_SERVER_0_CLIENT_1_0`) liefert das Trigger/Zustand-Bündel über `TRIGGER.TRIGGER` an `SPLIT` (`AE_AX_AX_SPLIT`), das an `FLIPFLOP.CLK` (`AE_AX_T_FF`) weiterreicht; `SPLIT.AX_OUT` schaltet parallel `DigitalOutput_Q1`.

Technische Besonderheiten

- **`AE_AX_*` als einwertige Basis-Kombi**: Die einfachste der drei Kombi-Familien in dieser Serie — ein einzelner `TRIGGER`-Pin statt `S_R` (Training_07) oder `S_R_T` (Training_06), da ein reiner Toggle-Trigger keine Aktionsunterscheidung braucht.
- **Vollständiger Adapter-Baukasten sichtbar**: Zusammen mit Training_06/07 zeigt Training_09 alle drei Stufen der adapter-3.0.0-Kombi-Familie (`AE_AX_*`/`ASR_AX_*`/`ASRT_AX_*`) im direkten Funktionsvergleich zu ihren manuell verdrahteten Gegenstücken.

Lernziele

- Einwertige Adapter-Kombi-Bausteine (`AE_AX_CLIENT_0_SUBSCRIBE_1`, `AE_AX_SERVER_0_CLIENT_1_0`) als einfachste Stufe der Serie.
- Vollständiger Überblick: alle drei Adapter-Kombi-Stufen (`AE`/`ASR`/ `ASRT`) im Vergleich zu ihren manuellen Gegenstücken.

Schwierigkeitsgrad: Fortgeschritten **Vorkenntnisse**: `Training_08_OPC_UA_RES`, `Training_06_OPC_UA_RES`/ `Training_07_OPC_UA_RES` (gleiches Adapter-Muster, andere Aktionsmengen).

Zusammenfassung

`Training_09_OPC_UA_RES` schließt die Serie ab: die einwertige `AE_AX_*`-Adapter-Variante des Toggle-only-Musters, funktional identisch zu Training_03/08, mit derselben Bündelungslogik wie Training_06/07.

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

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

2.3.3.11 Training 10 OPC UA RES

Training_10_OPC_UA_RES: Hoch/Runter-Schalter über OPC-UA — zwei Taster mit Interlock und Schutzzeit



• – ◦ • – ◦ • – ◦ •

Einleitung

`Training_10_OPC_UA_RES` ist eine verteilte Hoch/Runter-Schaltung: Gerät A liest die beiden physischen Taster `Input_I1` (Hoch) und `Input_I2` (Runter) und schreibt beide aktiv per OPC-UA (`AX_CLIENT_1_0`) auf die von Gerät B überwachten Knoten. Gerät B führt beide Zustände in `ILOCK_SWITCH_PROTECT_AX`

(Last-Wins mit Schutzzeit) zusammen und schaltet Output_Q7 und Output_Q8. Beide Taster sind tastend (spring-return); das Loslassen meldet sich als eigenes FALSE-Event über logiBUS_IXA.

Das Protokoll steckt im Composite (SUB style), nicht in der Resource des Geräts. Dies ist die erste Übung der Dreierserie Training_10–12 (Training_11 ersetzt die Taster durch VT-SoftKeys, Training_12 bündelt die Transportschicht).

Systemaufbau

Gerät A (FORTE_PC_A, MGR_ID = PC_A, Station 11, 192.168.1.11) mit EMB_RES_A, Gerät B (FORTE_PC_B, MGR_ID = PC_B, Station 12, 192.168.1.12) mit EMB_RES_B, verbunden über das Segment Ethernet. Jede Resource enthält nur E_TRIG (EVENTTYPE = 'EInit'), das von START.COLD angestoßen wird; die gesamte Logik steckt in den SubApps.

Verwendete SubApps

SubApp	Gerät	Inhalt
SubApp_PC_A	A (192.168.1.11)	Zwei Instanzen von logiBUS_IXA_TO_CLIENT_OPC (UpTaster: Input = Input_I1, ID = SW1_UP_REMOTE_WRITE; DownTaster: Input = Input_I2, ID = SW1_DOWN_REMOTE_WRITE)
SubApp_PC_B	B (192.168.1.12)	2x AX_SUBSCRIBE_1, ILOCK_SWITCH_PROTECT_AX, E_TimeOut, 2x logiBUS_QXA

OPC-UA-Adressraum

Die IDs stammen aus myOpcUaAddresses.gcf; die Stationen PC_A/PC_B stehen nur dort, die OPC-UA-IDs werden mit den Functions der Lib MyLib_OPC berechnet. Aufbau und Berechnung: OPC-UA-Adressen.

Konstanten (Schreiben auf anderer Station / lokal lesen)	Knoten	Bedeutung
SW1_UP_REMOTE_WRITE / SW1_UP_LOCAL_READ	/Objects/ Switch1/Up,1:s=Switch1_Up	Knoten auf B; A schreibt (WRITE), B liest lokal (READ)
SW1_DOWN_REMOTE_WRITE / SW1_DOWN_LOCAL_READ	/Objects/Switch1/ Down,1:s=Switch1_Down	Knoten auf B; A schreibt (WRITE), B liest lokal (READ)

Programmablauf und Verbindungen

1. **SubApp_PC_A** (Gerät A): UpTaster liest Input_I1, DownTaster liest Input_I2; beide (logiBUS_IXA_TO_CLIENT_OPC) schreiben den Zustand per OPC-UA-WRITE auf den Knoten von Gerät B (SW1_UP_REMOTE_WRITE bzw. SW1_DOWN_REMOTE_WRITE). Es gibt keine Verbindungen innerhalb der SubApp.
2. **SubApp_PC_B** (Gerät B), Adapterverbindungen:
   a. SUBSCRIBE_UP.OUT ⟶ ILOCK_AX.UP_IN
   b. SUBSCRIBE_DOWN.OUT ⟶ ILOCK_AX.DOWN_IN
   c. ILOCK_AX.timeOut ⟶ E_TimeOut.TimeOutSocket
   d. ILOCK_AX.UP_OUT ⟶ DigitalOutput_Q7.OUT
   e. ILOCK_AX.DOWN_OUT ⟶ DigitalOutput_Q8.OUT

Parameter: SUBSCRIBE_UP/SUBSCRIBE_DOWN (AX_SUBSCRIBE_1): QI = TRUE, ID = SW1_UP_LOCAL_READ bzw. SW1_DOWN_LOCAL_READ; ILOCK_AX (ILOCK_SWITCH_PROTECT_AX): DT_PROTECT = T#1s; DigitalOutput_Q7/ DigitalOutput_Q8 (logiBUS_QXA): QI = TRUE, Output = Output_Q7 bzw. Output_Q8.

Technische Besonderheiten

- **Kein Zwischenbaustein nötig**: AX_SUBSCRIBE_1.OUT ist bereits vom Adaptertyp AX — demselben Typ, den ILOCK_SWITCH_PROTECT_AX.UP_IN/.DOWN_IN als Socket erwarten.
- **Last-Wins mit Schutzzeit**: ILOCK_SWITCH_PROTECT_AX arbitriert Hoch und Runter; DT_PROTECT = T#1s legt die Schutzzeit fest, den Timer stellt E_TimeOut über den timeOut-Socket.
- **FB-Netz direkt im Composite**: Die Kombination (2x Subscribe + Interlock + Schutzzeit-Timer + 2x Ausgang) ist übungsspezifisch und steckt deshalb direkt in SubApp_PC_B.

- **Tastende Taster**: Das Loslassen eines Tasters wird als eigenes FALSE-Event über `logiBUS_IXA` gemeldet.

Lernziele

- Verteilte Hoch/Runter-Schaltung: Gerät A schreibt aktiv (`AX_CLIENT_1_0`), Gerät B überwacht lokal (`AX_SUBSCRIBE_1`).
- Zwei Adapter-Sockets (`AX`) direkt in `ILOCK_SWITCH_PROTECT_AX` führen (Last-Wins, Schutzzeit `DT_PROTECT`).
- Das Muster „SUB style“: Protokoll im Composite statt in der Resource.

Schwierigkeitsgrad: Fortgeschritten
Empfohlene Reihenfolge: `Training_09_OPC_UA_RES`.

Zusammenfassung

`Training_10_OPC_UA_RES` zeigt die Grundform der Serie: zwei physische, tastende Taster auf Gerät A, einzelne Netzwerk-Bausteine (`AX_CLIENT_1_0` in den Composites, 2x `AX_SUBSCRIBE_1`) und der Interlock mit Schutzzeit auf Gerät B.

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

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

2.3.3.12 Training 11 OPC UA RES

Training_11_OPC_UA_RES: Hoch/Runter-Schalter über OPC-UA — VT-SoftKeys mit Rückmeldung



• – ◦ • – ◦ • – ◦ •

Einleitung

`Training_11_OPC_UA_RES` ersetzt gegenüber `Training_10_OPC_UA_RES` die physischen `logiBUS_IXA`-Eingänge durch zwei VT-SoftKeys (`Softkey_IXA`, `SoftKey_F1` = Hoch, `SoftKey_F2` = Runter). Gerät A schreibt sie aktiv per OPC-UA (`CLIENT_1_0`) auf die von Gerät B überwachten Knoten. Gerät B ist identisch zu Training_10 (2x `AX_SUBSCRIBE_1` ⟶ `ILOCK_SWITCH_PROTECT_AX` ⟶ 2x `logiBUS_QXA`) und schreibt zusätzlich den tatsächlich erreichten Zustand (`UP_OUT`/`DOWN_OUT` nach Arbitrierung/Schutzzeit) per `AX_SPLIT_2` + `AX_CLIENT_1_0` zurück an Gerät A.

Auf Gerät A färbt je ein `GreenWhiteBackground1_AX` den jeweiligen SoftKey nach diesem zurückgemeldeten Zustand — also nicht nach dem rohen Tastendruck. `Softkey_IXA` liest kontinuierlich (wie `logiBUS_IXA`), ein REQ-Polling ist nicht nötig; `INIT` (`EInit`) startet die Überwachung automatisch.

Systemaufbau

Gerät A (FORTE_PC_A, MGR_ID = PC_A, Station 11, 192.168.1.11) mit EMB_RES_A, Gerät B (FORTE_PC_B, MGR_ID = PC_B, Station 12, 192.168.1.12) mit EMB_RES_B, verbunden über das Segment Ethernet. Jede Resource enthält nur E_TRIG (EVENTTYPE = 'EInit'), das von START.COLD angestoßen wird; die gesamte Logik steckt in den SubApps.

Verwendete SubApps

SubApp	Gerät	Inhalt
SubApp_PC_A	A (192.168.1.11)	2x Softkey_IXA (UpKey: SoftKey_F1; DownKey: SoftKey_F2), 2x AX_CLIENT_1_0 (Schreiben), 2x AX_SUBSCRIBE_1 (Rückmeldung lesen), 2x GreenWhiteBackground1_AX (BG_Up, BG_Down)
SubApp_PC_B	B (192.168.1.12)	2x AX_SUBSCRIBE_1, ILOCK_SWITCH_PROTECT_AX, E_TimeOut, 2x AX_SPLIT_2, 2x logiBUS_QXA, 2x AX_CLIENT_1_0 (Rückmeldung schreiben)

OPC-UA-Adressraum

Die IDs stammen aus myOpcUaAddresses.gcf; die Stationen PC_A/PC_B stehen nur dort, die OPC-UA-IDs werden mit den Functions der Lib MyLib_OPC berechnet. Aufbau und Berechnung: OPC-UA-Adressen.

Konstanten (Schreiben auf anderer Station / lokal lesen)	Knoten	Bedeutung
SW1_UP_REMOTE_WRITE / SW1_UP_LOCAL_READ	/Objects/Switch1/Up,1:s=Switch1_Up	Knoten auf B; A schreibt, B liest lokal
SW1_DOWN_REMOTE_WRITE / SW1_DOWN_LOCAL_READ	/Objects/Switch1/Down,1:s=Switch1_Down	Knoten auf B; A schreibt, B liest lokal
SW1_UP_STATE_REMOTE_WRITE / SW1_UP_STATE_LOCAL_READ	/Objects/Switch1/UpState,1:s=Switch1_UpState	Knoten auf A; B schreibt, A liest lokal
SW1_DOWN_STATE_REMOTE_WRITE / SW1_DOWN_STATE_LOCAL_READ	/Objects/Switch1/DownState,1:s=Switch1_DownState	Knoten auf A; B schreibt, A liest lokal

Programmablauf und Verbindungen

1. **SubApp_PC_A** (Gerät A), Adapterverbindungen:
   a. UpKey.IN ⟶ UpTasterClient.IN
   b. DownKey.IN ⟶ DownTasterClient.IN
   c. StateUpSubscribe.OUT ⟶ BG_Up.DI1
   d. StateDownSubscribe.OUT ⟶ BG_Down.DI1
2. **SubApp_PC_B** (Gerät B), Adapterverbindungen:
   a. SUBSCRIBE_UP.OUT ⟶ ILOCK_AX.UP_IN
   b. SUBSCRIBE_DOWN.OUT ⟶ ILOCK_AX.DOWN_IN
   c. ILOCK_AX.timeOut ⟶ E_TimeOut.TimeOutSocket
   d. ILOCK_AX.UP_OUT ⟶ SPLIT_UP.IN
   e. SPLIT_UP.OUT1 ⟶ DigitalOutput_Q7.OUT
   f. SPLIT_UP.OUT2 ⟶ StateUpClient.IN
   g. ILOCK_AX.DOWN_OUT ⟶ SPLIT_DOWN.IN
   h. SPLIT_DOWN.OUT1 ⟶ DigitalOutput_Q8.OUT
   i. SPLIT_DOWN.OUT2 ⟶ StateDownClient.IN

Parameter: Gerät A: UpKey/DownKey (Softkey_IXA): QI = TRUE, u16ObjId = SoftKey_F1 bzw. SoftKey_F2; UpTasterClient/DownTasterClient (AX_CLIENT_1_0): QI = TRUE, ID = SW1_UP_REMOTE_WRITE bzw. SW1_DOWN_REMOTE_WRITE; StateUpSubscribe/StateDownSubscribe (AX_SUBSCRIBE_1): QI = TRUE, ID = SW1_UP_STATE_LOCAL_READ bzw. SW1_DOWN_STATE_LOCAL_READ; BG_Up/BG_Down (GreenWhiteBackground1_AX): u16ObjId = SoftKey_F1 bzw. SoftKey_F2. Gerät B: SUBSCRIBE_UP/SUBSCRIBE_DOWN: QI = TRUE, ID = SW1_UP_LOCAL_READ bzw. SW1_DOWN_LOCAL_READ; ILOCK_AX:

`DT_PROTECT = T#1s; DigitalOutput_Q7/DigitalOutput_Q8: QI = TRUE, Output = Output_Q7` bzw. `Output_Q8; StateUpClient/StateDownClient: QI = TRUE, ID = SW1_UP_STATE_REMOTE_WRITE` bzw. `SW1_DOWN_STATE_REMOTE_WRITE`.

Technische Besonderheiten

- **Rückmeldung statt Tastendruck**: Die SoftKeys werden nach dem von Gerät B tatsächlich erreichten Zustand (nach ILOCK-Arbitrierung/Schutzzeit) gefärbt, nicht nach dem rohen Tastendruck.
- **State-Write von B nach A**: Gleiches Prinzip wie im Vorbild `SoftKeySR_PC_A_OPC` — Gerät B schreibt per `AX_CLIENT_1_0` auf einen lokal von Gerät A überwachten Knoten (`*_STATE_*`).
- **Verteilung per `AX_SPLIT_2`**: Der interlockte Zustand geht parallel an den physischen Ausgang und an den Rückmelde-Client.
- **Kontinuierliches Lesen**: `Softkey_IXA` braucht kein REQ-Polling (Vorbild `Softkey_IXA_TO_logiBUS_QXA_BG`).

Lernziele

- VT-SoftKeys (`Softkey_IXA`) als OPC-UA-Eingabe statt physischer Eingänge.
- Bestätigten Zustand zurückschreiben (B ⟶ A) und per `GreenWhiteBackground1_AX` anzeigen.
- Interlock-Ausgang per `AX_SPLIT_2` auf Ausgang und Rückmeldung verteilen.

Schwierigkeitsgrad: Fortgeschritten
Empfohlene Reihenfolge: `Training_10_OPC_UA_RES`.

Zusammenfassung

`Training_11_OPC_UA_RES` erweitert Training_10 um VT-SoftKeys und eine Rückmeldung: Gerät B meldet den nach Interlock und Schutzzeit tatsächlich erreichten Zustand an Gerät A zurück, wo er den SoftKey einfärbt.

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

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

2.3.3.13 Training 12 OPC UA RES

Training_12_OPC_UA_RES: Hoch/Runter-Schalter über OPC-UA — gebündelte A2X-Transportschicht



• – ◦ • – ◦ • – ◦ •

Einleitung

`Training_12_OPC_UA_RES` entspricht funktional `Training_11_OPC_UA_RES`, bündelt aber die Transportschicht: Statt je 2x `AX_SUBSCRIBE_1`/`AX_CLIENT_1_0` (4 einzelne Netzwerk-Bausteine, 4 Konstanten) über-

nimmt **ein** Baustein den kompletten Roundtrip für Hoch und Runter (je ein `CLIENT_2_0/SUBSCRIBE_2`, 2 Werte pro Aufruf, 2 Konstanten).

- Gerät A: die 4 lokalen Bausteine (2 SoftKeys + 2 `GreenWhiteBackground`) stecken gebündelt in `Softkey2_TO_A2X2_BG`, das nach außen nur einen einzigen `A2X2`-Plug zeigt; dieser wird direkt in `A2X2_CLIENT_2_0_SUBSCRIBE_2` gesteckt.
- Gerät B: `A2X2_CLIENT_2_0_SUBSCRIBE_2_PLUG` übernimmt den Roundtrip. `A2X2_ILOCK_SWITCH_PROTECT` kapselt Bridge (`A2X2_TO_A2X`), Interlock (`ILOCK_SWITCH_PROTECT_A2X`) und Split (`A2X_SPLIT_2`) in einem Composite (`DT_PROTECT` als Parameter) und liefert den interlockten Zustand als `A2X`-Plug. `A2X_TO_QXA2` entbündelt auf die 2 physischen `logiBUS_QXA`-Ausgänge.

Die Daten bleiben somit `A2X`-gebündelt bis unmittelbar vor der physischen Hardware.

Systemaufbau

Gerät A (`FORTE_PC_A`, `MGR_ID = PC_A`, Station 11, `192.168.1.11`) mit `EMB_RES_A`, Gerät B (`FORTE_PC_B`, `MGR_ID = PC_B`, Station 12, `192.168.1.12`) mit `EMB_RES_B`, verbunden über das Segment `Ethernet`. Jede Resource enthält nur `E_TRIG` (`EVENTTYPE = 'EInit'`), das von `START.COLD` angestoßen wird; die gesamte Logik steckt in den SubApps.

Verwendete SubApps

SubApp	Gerät	Inhalt
`SubApp_PC_A`	A (`192.168.1.11`)	`Keys` (`Softkey2_TO_A2X2_BG`: `u16ObjId_UP = SoftKey_F1`, `u16ObjId_DOWN = SoftKey_F2`), `NET` (`A2X2_CLIENT_2_0_SUBSCRIBE_2`: `QI = TRUE`, `ID_WRITE = SW2_PRESS_REMOTE_WRITE`, `ID_READ = SW2_STATE_LOCAL_READ`)
`SubApp_PC_B`	B (`192.168.1.12`)	`NET` (`A2X2_CLIENT_2_0_SUBSCRIBE_2_PLUG`: `QI = TRUE`, `ID_WRITE = SW2_STATE_REMOTE_WRITE`, `ID_READ = SW2_PRESS_LOCAL_READ`), `ILOCK` (`A2X2_ILOCK_SWITCH_PROTECT`: `DT_PROTECT = T#1s`), `Outputs` (`A2X_TO_QXA2`: `Output_UP = Output_Q7`, `Output_DOWN = Output_Q8`)

OPC-UA-Adressraum

Die IDs stammen aus `myOpcUaAddresses.gcf`; die Stationen `PC_A`/`PC_B` stehen nur dort, die OPC-UA-IDs werden mit den Functions der Lib `MyLib_OPC` berechnet. Aufbau und Berechnung: OPC-UA-Adressen.

Konstanten (Schreiben auf anderer Station / lokal lesen)	Knoten	Bedeutung
`SW2_PRESS_REMOTE_WRITE` / `SW2_PRESS_LOCAL_READ`	`/Objects/Switch2/Up,1:s=Switch2_Up;/Objects/Switch2/Down,1:s=Switch2_Down`	Zwei Knoten in einem String (`;`), auf B; A schreibt, B liest lokal
`SW2_STATE_REMOTE_WRITE` / `SW2_STATE_LOCAL_READ`	`/Objects/Switch2/UpState,1:s=Switch2_UpState;/Objects/Switch2/DownState,1:s=Switch2_DownState`	Zwei Knoten in einem String (`;`), auf A; B schreibt, A liest lokal

Programmablauf und Verbindungen

1. **`SubApp_PC_A`** (Gerät A), Adapterverbindung:
   a. `Keys.OUT` ⟶ `NET.IO`
2. **`SubApp_PC_B`** (Gerät B), Adapterverbindungen:
   a. `NET.IO` ⟶ `ILOCK.IO`
   b. `ILOCK.OUT` ⟶ `Outputs.IN`

Technische Besonderheiten

- **Zwei Konstanten statt vier**: `SW2_*` bündelt zwei Knoten in einem String (mit `;` getrennt) für `SUBSCRIBE_2`/`CLIENT_2_0` (siehe OPC-UA-Adressen).

- **A2X-Bündelung bis zur Hardware**: Auf Gerät B bleibt die Kette Bridge → Interlock → Split in `A2X2_ILOCK_SWITCH_PROTECT` gekapselt; erst `A2X_TO_QXA2` entbündelt auf die physischen Ausgänge.
- **`SUBSCRIBE_2` statt Methodenaufruf**: Ein einfacher OPC-UA-Methodenaufruf (`CLIENT_2`/`SERVER_2`, Request/Response) wurde für den gebündelten Roundtrip verworfen — der Remote-Zustand kann sich unabhängig von den eigenen Schreibzugriffen aus einer dritten Quelle ändern, und die lokale Oberfläche muss auf diesen Push reagieren, was nur `SUBSCRIBE_2` leistet (Begründung im Commit `04b465d77`).
- **Plug und Socket**: Gerät A nutzt den Socket-Baustein `A2X2_CLIENT_2_0_SUBSCRIBE_2`, Gerät B die Plug-Variante `A2X2_CLIENT_2_0_SUBSCRIBE_2_PLUG`.

Lernziele

- Mehrere Werte pro OPC-UA-Aufruf bündeln (`A2X2_CLIENT_2_0_SUBSCRIBE_2`, 2 Konstanten statt 4).
- Bündelnde Composites (`Softkey2_TO_A2X2_BG`, `A2X2_ILOCK_SWITCH_PROTECT`, `A2X_TO_QXA2`) und die A2X-Variante des Interlocks (`ILOCK_SWITCH_PROTECT_A2X`) einsetzen.
- Unterschied Plug-/Socket-Variante des Netzwerkbausteins verstehen.

Schwierigkeitsgrad: Fortgeschritten
Empfohlene Reihenfolge: `Training_11_OPC_UA_RES`.

Zusammenfassung

`Training_12_OPC_UA_RES` ist die gebündelte Variante von Training_11: dieselbe Funktion, aber ein einziger `A2X2`-Roundtrip je Gerät, `A2X`-gebündelt bis unmittelbar vor SoftKeys bzw. physischen Ausgängen.

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

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

2.3.4 Numerischer Index (test_VV)

Hier finden Sie alle Übungen des Arbeitsbereichs `test_VV` in aufsteigender Reihenfolge.

- Training_01_PUBLISH_SUBSCRIBE
- Training_01_CLIENT_SERVER
- Training_02_CLIENT_SERVER2
- Training_01_OPC_UA_SUB
- Training_02_OPC_UA_RES
- Training_03_OPC_UA_RES
- Training_04_OPC_UA_RES
- Training_05_OPC_UA_RES
- Training_06_OPC_UA_RES
- Training_07_OPC_UA_RES
- Training_08_OPC_UA_RES
- Training_09_OPC_UA_RES

2.3.4.1 🎓 training1 - Uebungen VV (Verteilte Verarbeitung)

🎧 Podcast

- 4diac IDE: Dein Open-Source-Werkzeugkasten für verteilte Industrieautomatisierung nach IEC 61499
- IEC 61499: Die Zukunft der industriellen Automatisierung und verteilten Systeme
- Von der SPS zur verteilten Intelligenz: IEC 61499 für Industrie 4.0 verstehen
- DIN EN 61499-1 Entmystifiziert: Funktionsbausteine, Objektorientierung und verteilte Systeme
- DIN EN 61499-1 Entschlüsselt: Der Bauplan für modulare, verteilte Steuerungssysteme

📺 Video

- Der denkende Traktor
- Der Traktor-Vordenker
- Die Kunst des Lötens
- EAGLE CAD: A Legend
- Ideale Dioden

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

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

3. static

3.1 Fonts

3.1.1 TGL Normschrift – _std Font Variants

3.1.1.1 Overview

The `_std` variants of the TGL Normschrift fonts fix the **U+0023 (#)** glyph, which in the original fonts incorrectly contains a „m/m“ (millimeters) symbol instead of the standard number sign / hash character.

This makes the fonts usable in contexts where a proper # is required (e.g. programming, technical documentation, IEC 61499 function block diagrams).

3.1.1.2 Fonts

File	Style	Based on
`TGL 0-16_std.ttf`	Italic (75° slant)	TGL 0-16
`TGL 0-17_std.ttf`	Upright	TGL 0-17
`TGL 0-17 alt_std.ttf`	Upright (alternate)	TGL 0-17 Alt

3.1.1.3 Changes

New glyph at U+0023 (#)

A number sign was designed from scratch to match each font's style:

- **Stroke width** matches the font (207–208 units)
- **Round stroke caps** using the same quadratic Bézier curve pattern as the original glyphs (I, l, hyphen)
- **Height** is approximately 2/3 of the capital letter height, consistent with typical number sign proportions
- **TGL 0-16_std** (italic): vertical strokes are slanted at 75° from the baseline, matching the font's standard lean angle per DIN/TGL specification for slanted Normschrift
- **TGL 0-17_std / TGL 0-17 alt_std** (upright): all strokes are perpendicular to each other
- The glyph is constructed as a single non-overlapping outer contour with one inner counter (hole), avoiding rendering artifacts from overlapping paths

Original m/m glyph preserved at U+E000

The original „m/m“ symbol that previously occupied the # position has been moved to **U+E000** (Unicode Private Use Area) so it remains accessible if needed.

Internal font names

All internal name table entries (family name, full name, PostScript name, unique ID) have `_std` appended to avoid conflicts when both original and modified fonts are installed on the same system.

3.1.1.4 Technical Details

- Format: TrueType (.ttf)
- Units per em: 2048
- Modified using Python `fonttools` library
- No other glyphs were changed
- All original font metrics, hinting, and licensing information are preserved

3.1.1.5 License

The original TGL Normschrift fonts are Copyright (c) 2010 by Peter Wiegel, released under GPL with font exception and OFL (Open Font License). The `_std` modifications are distributed under the same terms.